Háromrétegű tervezés

Ismerjük meg a köztes programréteg kialakítását a mod_perl/Apache környezetben.

Néhány hónapja megvizsgáltuk a Masont (Linuxvilág 2000. november, 59. oldal), mely egy korszerű webfejlesztő eszköz. Ez a mod_perlt, az Apache-t és a sablonokat használja. Az egyik példánk egy sajtóközleményeket kezelő rendszer volt, amelyben Mason-elemek szedik elő a legfrissebb sajtóközleményeket egy adatbázisból. Ebben a cikkben az úgynevezett kétrétegű programozói stílusra ismerhetünk rá, azaz a Mason-programok közvetlenül beszélgetnek az adatbázissal a Perl DBI és mod_perl Apache::DBI használatával.

De ahogy sokan megjegyezték a Mason levelezőlistán, ez a megközelítés – amelyben az SQL utasítások közvetlenül a Mason összetevőkben szerepelnek – sokszor nem célszerű. Az adatbázis szerkezetének változása, vagy egy másikfajta adatbázis-kezelőre való áttérés arra kényszerítene, hogy módosítsuk magukat az összetevőket. Ráadásul a nem webes programoknak újra meg kell valósítaniuk az SQL hívásokat a saját kódjukban ahelyett, hogy egy egységesen fenntartott és ellenőrzött közös programkönyvtárat használnának. Mindkét gond megoldódik, ha egy új réteget adunk a programhoz, amely az adatbázis és a Mason-programok között helyezkedik el. Ezt az egyre népszerűbb felépítést nevezik háromrétegű megközelítésnek, mivel három, egymástól jól elhatárolható programcsoporttal van dolgunk. Ezek az adatbázis, a középső réteg és a megjelenítési réteg (a felhasználói program).

Most és a következő hónapban egy egyszerű webalapú címjegyzéket és határidőnaplót fogunk vizsgálni, amelyen bemutatjuk a háromrétegű megközelítést. Remélem sikerül megvilágítani a módszer előnyeit és hátrányait, és mindenki képes lesz értékelni e módszer jelentőségét, ha egy nagyobb webhelyet kell kialakítania.

Miután áttekintettük az általános felépítést, készek leszünk arra, hogy megismerkedjünk a Java Server Pages és a Jakarta-Tomcat rejtelmeivel és az alkalmazáskiszolgálókkal. Megvizsgáljuk az e megközelítésben felbukkanó csapdahelyzeteket, és azt is, hogy miként segíti ez a gondolkodásmód a fejlesztés egyszerűbbé és méretezhetőbbé tételét hosszú távon.

Az adatbázisok

Az első, talán legfontosabb réteg a relációs adatbázis. Ebben a példában PostgreSQL-t fogok használni, de használhatnánk bármi mást is, mondjuk Oracle-t vagy MySQL-t.

Az adatbázis megtervezése előtt meg kell határoznunk, hogy mit akarunk tenni, azaz mire akarjuk használni. A cikk céljaira bőven elég egy rövid és homályos leírás az elérendő célokról: szeretnénk egy címjegyzéket, ami a Weben keresztül megnézhető, kereshető és frissíthető. Ezen felül szeretnénk a webböngésző segítségével a határidőnaplóba találkozókat bejegyezni, onnan törölni, illetve módosítani.

Ehhez legalább két tábla szükséges, az egyikben tartjuk nyilván az embereket, a másikban a találkozókat. Az *1. listán* olvasható az emberek adatait tartalmazó táblát létrehozó kód.

Az egyes emberek keresztnevét, vezetéknevét, országát és elektronikus levélcímét mindig tároljuk, ezenkívül pedig a címet, a hozzá kapcsolható várost, az államot, az irányítószámot és egy megjegyzést is készíthetünk. Ezzel feltételeztük, hogy minden embernek van levélcíme – olyan feltevés ez, ami egyre inkább igaz, de nem biztos, hogy jó ez a jellemző, ha számos barátunk és üzletfelünk dolgozik a számítástechnikai iparon kívül.

Az embereket tartalmazó tábla minden eleme egyértelműen azonosítható a `person_id` oszlop által, amelyet a PostgreSQL automatikusan növel. Biztosak lehetünk abban is, hogy minden ember csak egyszer szerepel a listán, mivel a levélcímnek egyedinek kell lennie.

Ez megengedi, hogy két Szabó István nevű barátunk legyen, de azt is jelenti, hogy egy közös levélcímet használó házaspár tagjait nem tudjuk külön-külön felvenni a listára. Szintén nehézkes a több levélcímet használó emberek kezelése.

Egy új ember felvétele a táblába viszonylag egyszerű:

```
INSERT INTO People
       (first_name, last_name, address1,
       address2, email, city, state, postal_code,
       country, comments)
       VALUES
       ('György', 'Szy', 'Népszínház u. 31',
       'I. em. 7.', 'szy@linuxvilag.hu',
       'Budapest', NULL, '1081', 'Magyarország',
       'A szerkesztőség címe')
       ;
```

Az oszlopok többségének alapértelmezett értéke a `NULL`, tehát egy ember felvétele elvégezhető úgy is, hogy kizárólag a kötelezően kitöltendő oszlopokba írunk:

1. lista A People tábla meghatározása

```
CREATE TABLE People (
       person_id   SERIAL       NOT NULL,
       first_name  VARCHAR(20)  NOT NULL  CHECK  (first_name <>''),
       last_name   VARCHAR(20)  NOT NULL  CHECK  (last_name <>''),
       address1    VARCHAR(30)  NULL      CHECK  (address1 <>''),
       address2    VARCHAR(30)  NULL      CHECK  (address2 <>''),
       email       VARCHAR(50)  NOT NULL  CHECK  (email ~'@'),
       city        VARCHAR(30)  NULL      CHECK  (city <>''),
       state       VARCHAR(2)   NULL      CHECK  (state <>''),
       postal_code VARCHAR(10)  NULL      CHECK  (postal_code <>''),
       country     VARCHAR(20)  NOT NULL  CHECK  (country <>''),
       comments    TEXT         NULL      CHECK  (comments <>''),
       PRIMARY KEY(person_id),
       UNIQUE(email)
    );
```

```
INSERT INTO People
       (first_name, last_name, email, country)
       VALUES
       ('János', 'Kiss', 'janos@kiss.hu',
       'Magyarország')
       ;
```

Készítsük el most a találkozókat tartalmazó táblát. Ebben a People táblában tárolt személyekkel tervezett találkozóinkat tároljuk:

```
CREATE TABLE Appointments
       (
       person_id   INT4       NOT NULL
                              REFERENCES People,
       start_time  TIMESTAMP  NOT NULL,
       end_time    TIMESTAMP  NOT NULL,
       notes       TEXT       NULL
                              CHECK (notes <>''),
       UNIQUE(start_time)
       );
```

Most, hogy a találkozók táblája elkészült, hozzáadhatunk egy új találkozót úgy, hogy egy sort illesztünk a táblába:

```
INSERT INTO Appointments
       (person_id, start_time, end_time, notes)
       VALUES
       (1, 'January 22, 2001 14:00',
       'January 22, 2001 14:30', 'Tárgyalás')
       ;
```

Mivel a `person_id` a People táblából származó idegen kulcs, csak olyan emberrel szervezhetünk találkozót, aki megtalálható a People táblában. A mi céljainknak ez megfelel így is, de egy bonyolultabb, jobban megtervezett rendszer valószínűleg rugalmasabb lenne. Természetesen az adatbázis nem fogja hagyni, hogy egy időben több emberrel találkozzunk.

Középréteg

Miután megalkottuk a kiindulási adatbázist, gondolkozzunk el a középső réteg szerkezetén, amely elszigeteli egymástól az adatbázist és a webes alkalmazást. Ha egyszer egy másikfajta adatbázis-kezelőre váltunk, vagy az adatbázist ASCII vagy DBM fájlokra cseréljük, az objektumok rétege változatlan marad.
Ráadásul a nem webalapú alkalmazások is használhatják ezt a réteget arra, hogy az adatbázishoz hozzáférjenek, ezzel elérhetővé válik például a határidőnaplónk kimentése XML formátumban, vagy találkozók beolvasása más programokból.
Ezt a középső réteget szokás az alkalmazás ügymenetének, vagy üzleti logikájának (business logic) is nevezni. Az adatbázis lehetővé teszi, hogy adatainkat könnyen, gyorsan tároljuk és elérjük.
A Mason-összetevők segítségével könnyűszerrel készíthetünk dinamikus kimenetet a végfelhasználó számára. A középréteg megpróbálja rákényszeríteni az adatbázist, hogy minél több számítást végezzen el a beépített függvények, a nézetek és a tárolt eljárások használatával. Az alkalmazás működését meghatározó logika azonban a középső rétegben helyezkedik el.
A Perl legalább két lehetőséget kínál e réteg létrehozására. Az egyik lehetőség, hogy egy egyszerű Perl-modult hozunk létre, ennek függvényei és változói segítségével a feladat megoldható. Az ilyen

3. lista Retrieve-people.pl

```
#!/usr/bin/perl -w
use strict;
use People;
# Új People példány létrehozása
my $people = new People;
# A keresett személy kiválasztása név szerint
$people->set_current_person_by_name("Shai",
                                    "Re'em")
    || die "Hiba a személy kiválasztásakor";
# A személy adatainak lekérése
my $info = $people->get_current_info();
# Az adatok kiíratása
foreach my $key (sort keys %{$info})
{
    if (defined $info->{$key})
 {
        print "$key => $info->{$key}\n";
    }
}
print "-" x 60, "\n";
# ----------------------------------------------
# Új személy beszúrása
my $success = $people->new_person(
                  first_name => "Reuven",
                  last_name => "Lerner",
                  country => "Israel",
                  email => 'reuven@lerner.co.il',
                  phone => '08-973-2225');

if ($success)
{
    # Adatok kérése erről a személyről
    $info = $people->get_current_info();
    # Az adatok kiíratása
    foreach my $key (sort keys %{$info})
    {
 if (defined $info->{$key})
        {
            print "$key => $info->{$key}\n";
        }
    }
}
else
{
    print "Hiba!\n";
}
exit;
# ----------------------------------------------
# Most állítsuk a nevet valami másra
$people->update_first_name("Yochai");
# A személyhez tartozó adatok lekérése
$info = $people->get_current_info();
# Az új adatok kiíratása
foreach my $key (sort keys %{$info})
{
    if (defined $info->{$key})
    {
        print "$key => $info->{$key}\n";
    }
}
print "\n";
```

eljárásalapú programozói felület könnyen megírható, és ugyanolyan gyorsan fut, mint a többi Perl-program. A Perl lehetővé teszi az objektumalapú modul megírását is.
Bár kissé nehezebb Perl-objektumokat írni, és az eljárásaik is lassabban futnak le, mint az egyszerű függvények, a segítségükkel könnyebb átlátni és megírni a programot.
Néhány komoly kérdésre meg kell találnunk a választ, mielőtt belevágnánk a középső réteg létrehozásába. Milyen objektumokat hozzunk létre? Készíthetnénk egy olyan adatbázis-objektumot, amely minden lekérdezést úgy kezel, hogy a megfelelő SQL paranccsá fordítja le őket. Igen ám, de esetenként csak az emberről szóló adat érdekel, az esetleg vele szervezett találkozók nem, tehát legalább két objektumra lesz szükség, egy People és egy Appointment objektumra. Mivel az adatbázisunkat úgy terveztük meg, hogy minden találkozóhoz hozzárendelődik egy és csak egy ember, a találkozó objektumát csak az ember objektuma után határozhatjuk meg.

People.pm

A People.pm (a *2. és 4. lista* túl hosszú ahhoz, hogy az újság hasábjain közöljük, de a cikkhez tartozó összes lista letölthető tgz formátumban az ➲ftp.ssc.com/pub/lj/listings/issue81/ címről.) Ez egy objektummodul, amely néhány egyszerű feladatot végez el az imént létrehozott People táblával. Az objektum nem teljes és néhány helyen lenne mit csiszolni rajta, de arra jó, hogy bemutassa egy relációs adatbázis elérését egy objektumalapú középső rétegen keresztül.
Az alapötlet az, hogy létrehozzuk a People objektum egy új példányát, és aztán a határidőnaplóban ezen objektum segítségével kezeljük az embereket. Az adatbázisban szereplő összes ember kigyűjtéséhez a `get_all_full_names` eljárást használhatjuk, mint ebben a kódrészletben is látható. (Lásd még az ➲ftp.ssc.com/pub/lj/listings/issue81/ címről letölthető *4. listát*):

```
use People;
# Egy People objektum létrehozása
my $people = new People;
# Az összes név kiolvasása
my @names = $people->get_all_full_names();
# A nevek kinyomtatása
foreach my $name (@names)
{
    print "name\n";
}
```

Egy adott emberre vonatkozó adat megszerzéséhez vagy módosításához először meg kell adni a szóban forgó személyt. Mivel a középső rétegnek az a lényege, hogy megszabadítsa a felhasználót a kulcsok és egyéb elsődleges azonosítók használatától, a kívánt személy kiválasztható lesz a keresztnév, a vezetéknév vagy a levélcím alapján.
A levélcím biztosan egyedi az adatbázis szintjén, ezért a `set_current_person_by_email` a legbiztonságosabb eljárás. Ennek ellenére gyakran hasznos az embereket keresztnevük és vezetéknevük alapján azonosítani, ezért használható a `set_current_person_by_name` eljárás is. A jelenlegi megvalósításban az eljárás az első olyan sort fogja visszaadni az adatbázisból, amelyre a megadott név megegyezik. Ez nem feltétlenül az lesz, akit mi szeretnénk visszakapni. Miután a program beállította az kívánt személyt, adatait a `get_current_info` eljárással olvashatjuk ki:

```
# A kívánt személy beállítása név szerint
$people->set_current_person_by_name
        ➥("János","Kiss")
|| die "Hiba: a személy nem található!";
# Az adatok nyomtatása
foreach my $key (sort keys %{$info})
{
       if (defined $info->{$key})
       {
              print "$key => $info->{$key}\n";
       }
}
```

A People objektum minden példánya két adatot tárol: a pillanatnyilag kiválasztott személy azonosítóját (`$self->{current_person}`) és az adatbázis-azonosítót, amely az adatbázishoz kapcsol minket (`$self->{dbh}`). Azért érdemes tárolni az adatbázis-azonosítót, mert az adatbázishoz kapcsolódás viszonylag drága művelet. Időt nyerhetünk, hogy az adatbázishoz kapcsolódást a konstruktorban (létrehozóban) valósítjuk meg, és mindig ezt a kapcsolatot használjuk, amikor az objektum egy eljárását hívjuk.
Ez természetesen azt jelenti, hogy az adatbázis-kapcsolatot a Perl-objektum életének végén meg kell szüntetni. Ez egy kicsit cseles, mert a Perlben nincs kifejezetten destruktor (megszüntető), lévén a Perl egy szemétgyűjtő nyelv. A megoldás az, hogy egy `DESTROY` nevű eljárást hozunk létre, ezt kell az objektum megszűnésekor meghívni. A mi `DESTROY` eljárásunk egyszerűen csak lezárja az adatbázis-kapcsolatot, ezáltal az objektumot nyugodtan törölhetjük, nem fog memóriaszivárgást okozni sem az adatbázis-kezelőben, sem az ügyfélprogramban:

```
sub DESTROY
{
       # Saját magunkra hivatkozás megszerzése
       my $self = shift;
       # Adatbázis-azonosító megszerzése
```

5. lista Insert-appointment.pl

```
#!/usr/bin/perl -w
use strict;
use People;
use Appointments;
# Új People példány létrehozása
my $people = new People;
# Új Appointments példány létrehozása
my $appointments = new Appointments;
# A kívánt személy kiválasztása név szerint
$people->set_current_person_by_name("Hadar",
                                    "Re'em")
    || die "Hiba: a személy nem található!";
# Új találkozó hozzáadása a kívánt személlyel
my $result = $appointments->add_appointment
                    ➥($people,
                    'October 22, 2000 18:30',
                    'October 22, 2000 19:00',
                    'Társasjáték');
# Sikerült?
if ($result)
{
    print "Sikerült!\n";
}
else
{
    print "Nem sikerült: '$DBI::errstr'"
                            ➥unless $result;
}
```

6. lista Print-appointments.pl

```
#!/usr/bin/perl -w
use strict;
use diagnostics;
use Appointments;
# Új Appointments példány létrehozása
my $appointments = new Appointments;
# Az aznapi találkozók listájának elkészítése
my @appointments = $appointments->get_today();
# Végimegy a találkozókon
foreach my $appointment (@appointments)
{
    # Mindegyik találkozó mutatótömb, tehát az
    # az elemeit kell kiíratni
    print $appointment->{start_time}, ":\t";
    print "\t", $appointment->{first_name},
        " ", $appointment->{last_name}, "\n";
 print "\t", $appointment->{notes}, "\n";
}
```

```
        my $dbh = $self->{dbh};
        # Az adatbázis-kapcsolat lezárása
        $dbh->disconnect;
        return;
}
```

A `new_person` eljárás segítségével akár új személyeket is bejegyezhetünk. Értékként meg kell adni a kulcskészletet és az értékkészletet, amelyeket felhasználva aztán a a középső réteg létrehozza a megfelelő SQL-kifejezést:

```
# Új ember hozzáadása
my $success = $people->new_person
        (first_name => "Reuven",
        last_name => "Lerner",
        country => "Israel"
        email => 'reuven@lerner.co.il',
        phone => '08-973-2225');
print "A hozzáadás sikerült." if $success;
```

Mivel a Perlben a meghatározatlan (`undef`) érték automatikusan az SQL `NULL` értékévé fordítódik, a kitöltetlen oszlopokba `NULL` kerül.

Találkozók

Most, hogy megvan az adatbázisunkban tárolt embereket kezelő osztály, létre kell hoznunk a találkozók osztályát. Egyelőre csak az új találkozók beillesztésével és az aznapi találkozók megjelenítésével foglalkozunk.
Az Appointment.pm (*4. lista* az ➲ftp.ssc.com/pub/lj/listings/issue81/ címen) szerkezete lényegében hasonló a People.pm felépítéséhez, a konstruktorban (létrehozóban) nyitja meg az adatbázis-kapcsolatot, és az automatikusan meghívott `DESTROY` eljárással zárja le. Ezenkívül az Appointment objektum nem tárol más állapotot, egyszerűen csak egy csővezetéket hoz létre az adatbázis felé, ennek segítségével új találkozókat hozhatunk létre és az aznapi találkozókat jeleníthetjük meg.
Például az *5. lista* egy rövid programot tartalmaz, amely az Appointments.pm-et használja egy új találkozó létrehozására. Létre kell hoznunk az emberek és a találkozók egy-egy példányát.

Miután megvan ez a két objektum, beállíthatjuk a megfelelő személyt. Ha a `set_current_person_by_name` az `undef` hibaüzenettel tér vissza, a program hibaüzenettel leáll.
Ha a kívánt személy beállítása sikerült, akkor létrehozhatunk vele egy találkozót. A dátum és az idő formátumát a PostgreSQL határozza meg (elég sok formátumot elfogad).
Hasonlóképpen olvashatjuk ki a mai napra előjegyzett találkozók listáját a *6. listán* bemutatott program használatával (print-appointments.pl).
A program a `get_today` eljárást használja, amely egy mutatótömb-listát (list of hash references) ad vissza. Megjegyzendő, hogy a `get_today` megvalósítása a DBI **fetchrow_hashref** eljárását használja, ami közismerten sokkal lassabb, mint a **fetchrow_arrayref**. Ennek ellenére az a megoldás sokkal kényelmesebbé teszi az életünket, mert így megvalósítható a `print-appointments` a *6. listán* bemutatott módon.
Végül megnézhetjük, hogy egy adott személlyel a mai napon mikor és hányszor találkozunk. A `get_today_with_person` eljárást erre használjuk. Természetesen ez azt jelenti, hogy előbb létre kell hoznunk a People egy példányát, és ki kell választanunk a kívánt személyt a fent tárgyalt eljárások egyikével. A `get_today_with_person` megvalósítása a felhasználó által átadott első értékként a People egy példányát várja, így a megfelelő személy kerülhet az SQL-lekérdezésbe. A *7. listán* olvasható program bemutatja, hogyan tudhatom meg az összes aznapi találkozóm időpontját, amelyeket az unokaöcsémmel, Shaival beszéltem meg.

Az objektumok megtervezése

Az objektumok középrétegben történő használatának egyik fő oka az, hogy egy elvonatkoztatott réteget nyújt, azaz amíg a csatolófelület jól meghatározott és állandó, a tényleges megvalósítás változhat.
Igaz viszont, hogy mint minden alapos programozási módszer, a jó objektumok megtervezése is nehéz feladat lehet. A Perl szabad

7. lista Print-appointments-with-shai.pl

```
#!/usr/bin/perl -w
use strict;
use diagnostics;
use People;
use Appointments;
# Új People példány létrehozása
my $people = new People;
# Új Appointments példány létrehozása
my $appointments = new Appointments;
# A kívánt személy kiválasztása név szerint
$people->set_current_person_by_name("Shai",
                                    "Re'em")
   || die "Hiba: a személy nem található!";
# Az aznapi találkozók listájának elkészítése
my @appointments =
  $appointments->get_today_with_person($people);
# Végimegy a találkozókon
foreach my $appointment (@appointments)
{
    # Mindegyik találkozó mutatótömb, tehát
    # az elemeket kell kiíratni
    print $appointment->{start_time}, ":\t";
    print "\t", $appointment->{first_name},
        " ", $appointment->{last_name}, "\n";
    print "\t", $appointment->{notes}, "\n";
}
```

Forrásművek

Sok könyv és cikk szól a háromrétegű tervezésről.

Például a John Wiley kiadásában Robert Orfali, Dan Harkey és Jeri Edwards: The Essential Distrubuted Objects Surviving Guide című könyve. Ez jó bevezető a háromrétegű szerkezetek elméletébe, valamint egyes kereskedelmi megvalósításokat is bemutat. Elmagyarázza a sokrétegű szerkezet fogalmait, ezek közül sok fontos akad a webes alkalmazások szempontjából. A mű egy kicsit elavult, az OpenDocot említi, de nem szól a DCOM-ról. Mindamellett jó áttekintést ad ezekről a programozási módszerekről.

Szintén a John Wiley kiadó gondozásában jelent meg Jeri Edwards és Deborah DeVoe 3-Tier Client/Server at Work című kézikönyve, amely az elmélet néhány jelenlegi megvalósítását tárgyalja.

A könyvek elolvasása után az ember azt hiheti, hogy minden feladatot a háromrétegű tervezéssel kell megoldani. Ez természetesen nem így van, a tervezés függ többek között attól is, hogy mit és hogyan kell megvalósítani. Egy jó esettanulmányt írt általában a webalapú alkalmazáskiszolgálókról és különösen a háromrétegű tervezésről Philip Greenspun (az ArsDigita alapítója és szerzője a kiváló Philip and Alex's Guide to Web Publishing című műnek, amit a Morgan-Kaufmann adott ki). Az esettanulmány a weben is olvasható a ➲ http://www.arsdigita.com/asj/application-servers.adp címen.

Végül, a Sun elkezdte a Javára terelni az emberek figyelmét, ami szerintük egy kiváló nyelv a háromrétegű megoldásokhoz. Ezt a következő hónapokban meg fogjuk vizsgálni. Erről olvashatunk pár dolgot a következő könyvek első fejezeteiben: Ed Roman: Mastering Enterprise JavaBeans, John Wiley kiadó, valamint Richard Monson-Haefel: Enterprise JavaBeans, O'Reilly and Associates kiadó.

hozzáférést enged az objektum belsejéhez, ez azonban azt a veszélyt hordozza magában, hogy jó API hiányában az objektummal dolgozó programozó kísértést érezhet arra, hogy belenyúljon az objektum belsejébe, és közvetlenül a megvalósítással dolgozzon. Ennek az lehet a következménye, hogy a program működésképtelenné válik, ha a megvalósítás változik, márpedig pont ezt a helyzetet akartuk megakadályozni az objektumok használatával.

Továbbá azt szeretnénk, hogy objektumaink megvalósításai viszonylag függetlenek legyenek egymástól. A People és az Appointment objektumok megvalósításakor nagy kísértést éreztem arra, hogy megengedjem a találkozóknak, hogy a szóban forgó személy azonosítószámát megszerezzék és használják. Természetesen ez megszegte volna azt szétválasztási határt, amelyet az objektumok létrehozásával felépítettem. A megoldás – ami bevallom, nem olyan elegáns, mint szerettem volna – a get_current_person eljárás megalkotása lett. Ezáltal az Appointment objektum úgy tudhatja meg a kívánt személy nevét, hogy nem kell tudni, hogy az honnan származik. Végül a `get_current_person` visszatérési értéke bekerül az SQL-kifejezésbe, és összehasonlításra kerül a People.person_id-vel, ez bizonyos értelemben mégiscsak megszegi a szétválasztást.

Végezetül figyeljük meg, hogy az egyes objektumok nem tárolnak állapotadatokat. Viszonylag egyszerű lenne például, hogy a People objektum a People tábla minden sorát kiolvassa, és elérhetővé tegye az őt hívó objektumok számára. Valóban, ez a megoldás jelentősen csökkentené az adatbázis felé irányuló forgalmat, és megengedné, hogy Perlben végezzük el az adatok kezelését ahelyett, hogy mindig az SQL-hez kelljen fordulnunk.

Csakhogy ez a megoldás több gondot okoz, mint ahányat megold. Például mi történne, ha két példányt hoznánk létre a People objektumból? Két objektumunk lenne, mindkettőben az emberek összes adata benne lenne. Ha az egyik objektum állapota változna, ez semmilyen módon nem jelentkezne a másik objektumnál. Még rosszabb esetben: mi történne, ha mindkét objektum állapota azelőtt változna, mielőtt a változások megjelennek az adatbázisban? Egy jó adatbáziskezelő valószínűleg képes megbirkózni ezekkel a versenyhelyzetekkel, de a Perl-objektumok nem. Továbbá mi lenne, ha a People tábla százezer személy adatait tartalmazná? Ennyi adat beolvasása az ügyfélprogramba memóriapazarlás lenne, és az adatbázis-kezelő nagy teljesítményű lekérdezési és adatkezelési eljárásai is kihasználatlanul maradnának.

Az objektumaink ezért csak az adatbázishoz kapcsolódó csővezetékek, amelyek lehetővé teszik, hogy a webalapú alkalmazásunk anélkül tudjon beszélgetni az adatbázissal, hogy SQL-kifejezéseket kelljen beágyazni, vagy ismernie kellene a táblázatok szerkezetét. Az objektumok által nyújtott szabványos API használatával elérhető, hogy az alatta fekvő megvalósítást megváltoztassuk anélkül, hogy ezeket a változásokat a világ tudomására hoznánk.

Összefoglalás

Megvizsgáltuk, hogy milyen okok vezetnek a háromrétegű rendszerek használatára, és tanulmányoztunk egy ezzel a módszerrel készült egyszerű alkalmazást. Látható, hogy máris képesek vagyunk kisebb alkalmazásokat létrehozni. A következő hónapban befejezzük a most elkezdett program megvalósítását, ennek a végeredménye egy egyszerű, mod_perl/Mason és PostgreSQL hármast használó határidőnapló-program csontváza lesz. Megtárgyaljuk a háromrétegű megoldások méretezhetőségével kapcsolatos gondokat és néhány csapdahelyzetet.

Reuven M. Lerner (reuven@lerner.co.il) cége internetes tanácsadást vállal, székhelyük Modi'in-ben, Izraelben van. A *Core Perl* című könyv szerzője, mely a Prentice Hall kiadónál jelenik meg. Szeretettel vár mindenkit az ATF honlapján ➲http://www.lerner.co.il/atf/.