Debreceni Egyetem

Informatikai Kar

ASPEKTUSORIENTÁLT PROGRAMOZÁSI RENDSZEREK

Témavezető:

Espák Miklós

Egyetemi tanársegéd

Készítette:

Kovács Judit

Programtervező matematikus

Debrecen
2009

Tartalomjegyzék

1. Bevezetés

1.1 Célkitűzés

A számítógépes programozás – kialakulása óta – hatalmas fejlődésen ment keresztül. Míg kezdetben a lyukkártyás módszer, valamint a gépi kód rendkívül időigényes, közvetlen, kézi megírása volt szükséges a viszonylag egyszerű programok elkészítéséhez, addig manapság egyre kevesebb időráfordítással, egyre összetettebb alkalmazások valósíthatók meg. Ezt a fordítóprogramok kialakulása, illetve az egyre újabb programozási nyelvek megjelenése tette lehetővé, mivel ily módon a programozónak mind kevesebb munkát kell elvégeznie egy adott feladat megvalósításához. A fejlődés során egyre magasabb szintű programozási nyelvek jelennek meg, és a szöveges nyelvek mellett már a grafikus programozási nyelvek is igen nagy teret hódítanak maguknak.

Diplomamunkám célja, hogy bemutassam ezen fejlődés egyik magas lépcsőfokát jelentő programozási technika, az aspektusorientált programozás kialakulását, valamint azon okokat, melyek szükségessé tették annak létrejöttét, illetve jól mutatják annak létjogosultságát. Szeretnék átfogó képet adni ezen programozási rendszerről, illetve bemutatni annak tényleges, fejlesztésbeli használatát. Ehhez egy saját kliens-szerver webalkalmazást kívánok elkészíteni, melyen jól láthatóvá válik az aspektusorientált programozás használatának előnye, illetve azok a követelmények, melyek teljesítéséhez elengedhetetlen ezen módszer használata.

2. Az aspektusorientált programozás

A számítástechnikában *aspektusorientált programozás* (AOP) [1] alatt egy programozási paradigmát értünk, ahol a másodlagos követelményeket megvalósító kód elkülönítésre kerül az elsődleges, többnyire üzleti logikát (angolul: business logic) megvalósító részektől. Célja legfőképp az átmetsző követelmények különválasztása, hogy ezzel növelje a modularitást, s így teremtse meg az *aspektusorientált szoftverfejlesztés* alapjait.

Maga az aspektusorientált szoftverfejlesztés összességében fejlesztői eszközöket, illetve programozási technikákat jelöl, melyek lehetővé teszik ezt a fajta követelmény-szétválasztást még a kód szintjén.

2.1 Az AOP kialakulása

A programfejlesztés folyamatos fejlődése során megfigyelhető, hogy időről-időre újabb és újabb programozási paradigmák, nyelvek jelennek meg, válnak elérhetővé a programfejlesztők számára. Ezek megjelenésének közös célja szinte minden esetben az előzőleg megjelent, elterjedté vált nyelvek hibáinak javítása, az újonnan felmerülő programozási problémák megoldhatóságának megkönnyítése – vagy éppen lehetségessé tétele –, az egy adott feladattípus megvalósításához szükséges idő lerövidítése, a kód újrafelhasználhatóságának megkönnyítése, valamint a kooperatív fejlesztés egyszerűbbé tétele.

Az AOP esete sem különbözik ettől, mivel megjelenését leginkább a napjainkban is elterjedtnek tekinthető *objektum-orientált* (OO) programozás hiányosságainak kiküszöbölése, illetve az annak specifikációi által támasztott korlátok legyőzhetősége tette szükségessé, lehetségessé.

Már a '70-es évek folyamán megfigyelhetőek voltak olyan törekvések, ötletek, melyek nagyban egybevágnak az aspektusorientált programozás célkitűzéseivel, vagy épp alapjaiul szolgálnak annak. Még 1968-ban, a THE operációs rendszeren dolgozva Dijkstra [2] teremtette meg az absztrakciós szint fogalmát, melynek célja az operációs rendszerek fejlesztésének megkönnyítése volt. Az absztrakciós szint k ($k > 0$) egy összetett szoftver-rendszer hierarchiájának egy olyan eleme, mely modulok olyan csoportjából áll, amik implementációja kizárólag k-1 szintű modulokat használ fel. A $k = 0$ szinten elhelyezkedő

elemek lehetnek ily módon a processzor utasításkészlete, vagy a használt programozási nyelv struktúrái, parancsai.

Szintén a '70-es években jelent meg egy másik igen jelentős fogalom Parnas [3] és Myers [4] munkásságának köszönhetően: a *modul*. A modul egy olyan szoftverkomponens, mely:

- explicit szintaktikai definícióval rendelkezik,
- explicit aktivációs metódussal (hívás, angolul: call) rendelkezik,
- szemantikailag független,
- egy jól definiált részfeladatot végez egy teljes szoftver rendszerben,
- jól definiált, de lehetőleg minimális csatlakozási ponttal rendelkezik a szoftver többi része felé, mind az adatfolyam (argumentumok, globális változók), mind a vezérlési folyamat (hívások a modulból, vagy épp a modulra) tekintetében
- implementációját nem kell ismerni a felhasználónak

A modul megteremtésével mindössze annyi szükséges egy felhasználó számára, hogy ismerje az adott modul interfészét: a modul nevét, annak argumentumait (számát, típusát), visszatérési értékeit, s bizonyos esetekben a lehetséges kivételeket, az ún. mellékhatásait (a modul használata által módosított globális változókat és egyéb objektumokat), s nem utolsó sorban a modul szemantikáját (funkcionalitását).

Mindkét esetben látható, hogy a szoftverfejlesztés lényeges elemét jelenti a problémák, feladatok definiálása, majd megfelelő módon történő elkülönítése és szervezése. Az aspektusorientált programozás egyik fontos eleme szintén a kód megfelelő szétválasztása, modularitásának növelése.

Habár gyökerei a '70-es évekre is visszanyúlnak, tényleges megjelenésről azonban Gregor Kiczales és csapatának [5] a Xerox PARC-nál a '90-es évek közepén végzett munkásságának eredményeképp beszélhetünk. Az általuk megteremtett első (és talán a mai napig egyben a legelterjedtebb is) AOP csomag, az AspectJ [6] 2001-ben került a nyilvánosság elé.

Az AOP mellett más törekvések is megjelentek, melyeknek szintén célja volt az átmetsző

követelmények szétválasztása. Így az IBM kutatócsapata is elkészítette saját rendszerét, s megjelentette a Hyper/J-t [7], mely a cég alphaWorks [8] weboldaláról érhető el. Az AspectJ-hez hasonlóan ez is egy Javához kapcsolódó kiegészítés, habár sokkal nehézkesebb ennek használata. A konfigurációs állományoknak már csak a megértése is igen nehéz feladat. Ezen praktikus használhatóságot megnehezítő ok is hozzájárulhatott ahhoz, hogy a Hyper/J projektet meg is szüntették, s a fejlesztők körében sem igazán terjedt el.

A kialakuláshoz vezető tényleges okok könnyebb megértéséhez, illetve szemléltetéséhez a következőkben egy valós példán keresztül kívánom röviden bemutatni az objektum-orientált programozást, valamint annak ilyen típusú problémáit.

2.1.1 Az OO programozás áttekintése

A jó szoftverfejlesztés lényeges, kulcsfontosságú eleme, hogy minden hozzáértő számára világos, jól átlátható kód jöjjön létre. Ezáltal a kód későbbi kiegészítése, módosítása, karbantartása, valamint a hibakeresés is könnyebbé válik. A későbbi fejlesztések megkönnyítése, illetve a fejlesztési folyamat egyszerűbbé tétele, lerövidítése miatt kulcs-szerepet játszik a kód modularizációja, a komponensek újrafelhasználhatósága. Emellett azonban a redundanciát kerülni kell.

A programfejlesztés fejlődésének ranglétráján az objektum-orientált programozás (OOP) megjelenése egy új lépcsőfokot jelent, s igyekszik megvalósítani a fent említett követelmények mindegyikét.

A programozási problémák objektum-központú megközelítése [9] alapjaiban változtatta meg a programozással kapcsolatos gondolkodásmódot. Mivel használata biztonságosabb, jobban áttekinthető programok elkészítését teszi lehetővé, s a kód újrafelhasználhatóságát is lehetővé teszi, rövid idő alatt, könnyen teret hódított magának a programozók körében.

Ezen megközelítés alapja a valós világ modellezése, ezért jobban közelíti a valóságot, a valós problémákat. A modellezés egyik alapeleme az osztály, mely egy absztrakt adattípus. Amint a valós világban a dolgoknak nem csak tulajdonságai vannak, de viselkedési jellemzői is, ekképp az osztály is rendelkezik attribútumokkal és módszerekkel. Az attribútumok az objektum – mint másik alapelem – tulajdonságait, a módszerek pedig viselkedését határozzák meg. Az objektum minden esetben egy előre definiált osztály példánya, annak

példányosításával jön létre. Habár egy adott osztály példányai mindig azonos tulajdonságokkal és viselkedési jellemzőkkel rendelkeznek, az egyes objektumok mégis egyediek. Az ilyen nyelvek egyik fontos jellemzője az egységbezárás. Ennek során az objektum állapotát jellemző attribútumokat és az ezeket megváltoztató vagy lekérdező metódusokat egy egységként kezelik, s elzárják azokat a külvilágtól. Így nincs szükség olyan, az objektumon belüli függvényekre, amelyek hozzáférnek más objektumhoz, illetve megváltoztathatják azokat, emellett nem kellenek olyan, objektumon kívüli függvények, melyek megváltoztathatják azt. Az objektumok példányosító osztályai meghatároznak egy interfészt a rendszer más objektumai számára, ami definiálja azokat a metódusokat, amelyeket más objektumok használhatnak az állapot megváltoztatására.

Az egyes objektumok kommunikációja üzenetküldés formájában történik. Az üzenetküldés általában a fogadó objektum egy látható, publikus metódusának meghívásával, illetve az objektum nevével történik. A rendszer ily módon tehát felfogható objektumok egy olyan csoportjának, ahol az objektumok üzeneteket küldenek egymásnak, melyek célja információk kérése vagy más objektumok megváltoztatása.

2.1.2 Az OO programozás problémái

Ugyan az objektumorientált paradigma nagy előrelépést jelentett a számítástechnika tudománya terén, mégis vannak olyan problémák, melyeket még az objektumorientált szemlélettel sem könnyű megoldani. Főleg nagyobb kiterjedésű rendszerek esetében jelent gondot az utólagos módosítás, ezért már a tervezés fázisában gondolni kell az utólagos továbbfejlesztési, kiegészíthetőségi igényekre. Emellett felléphetnek olyan követelmények is, melyek megvalósítása megsértheti az OOP egyik fontos jellemzőjét: az egységbezárást. Ilyen esetet jelenthet, ha egy objektumhoz felveszünk egy nem az ő feladatkörébe tartozó funkciót, ezért az szerves részévé válik az objektumnak, annak ellenére is, hogy a valóságban nem az.

Egy elkészítendő rendszer esetében mindig vannak követelmények, konkrétan amik miatt a rendszer el is készül, olyan funkciók, melyek működőképessége elengedhetetlen. Vannak azonban olyan követelmények is, melyek alapvetően nem a rendszer elsődleges funkcionalitását szolgálják, megvalósításuk azonban mégis lényeges. Ilyenek lehetnek például a naplózás vagy a biztonsági követelmények. A problémát gyakran ezek a követelmények okozzák, mivel megvalósításuk érdekében „bele kell nyúlnunk” az adott objektumokba.

Ezáltal az objektum már nem csak a saját, konkrét feladatát látja el, megsértve ezzel az egységbezárás elvét is. Ezek a követelmények továbbá több komponenst is érinthetnek, így szétszórtan helyezkednek el a program kódjában, s nem csak megnehezítik az utólagos karbantartást, az átláthatóságot, de összekuszált kódot is eredményeznek. Az osztályok módosítása és a nyomkövetés így nehézzé válik, a kód újrahasznosíthatósága is meghiúsul.

Egy tipikus rendszer – hogy a céljának megfeleljen –, számos ilyen követelménnyel kell rendelkezzen. A tervezőnek fel kell építenie a rendszert, mely a kívánalmakat teljesíti, ugyanakkor vigyáznia kell, hogy a használt módszertant ne sértse meg. Mivel a rendszerben megvalósítandó követelményeket az egyes osztályok implementálják, a cél az, hogy az egyes feladatokhoz tartozó kód elkülönüljön. Ez azonban sajnos nem mindig lehetséges. Tekintsük például a következő két követelményt:

1. A rendszernek nyilván kell tartania az adott termékek árát
2. Minden árváltozást nyilván kell tartani, és naplózni kell azokat

Az első követelmény alapján így készíthetünk egy `Product` osztályt, melynek van egy ár attribútuma, valamint az azt beállító és lekérdező metódusa. Ez egy absztrakt osztály, mely a rendszerben szereplő összes termék funkcionalitását kezeli:

```
public abstract class Product {
     private double price;
     Product() {
          price = 0.0;
     }
     public void putPrice(real p) {
          price = p;
     }
     public int getPrice() {
          return price;
     }
}
```

A második funkció implementálása sem jelent problémát, s önmagában nem kerül összetűzésbe az első követelménnyel. Így a `Logger` osztály:

```
public class Logger {
     private OutputStream ostream;
     Logger() {
          //open log file
     }
```

```
        void writeLog(String value) {
            //write value to log file
        }
    }
```

Az árváltozások naplózásához meg kell hívni a `writeLog()` metódust minden olyan helyen, ahol az ár változása megtörténik, ezért a `Product` osztály a következőképp változik:

```
public abstract class Product {
    private float price;
    Logger loggerObject;

    Product() {
        price = 0.0;
        loggerObject = new Logger();
    }

    public void putPrice(real p) {
        loggerObject.writeLog("Price changed from"
        + price + " to "+p);
        price = p;
    }

    public int getPrice() {
        return price;
    }
}
```

Az ár megváltoztatása a `putPrice()` metódus segítségével történik. Meghívásakor a naplózás is végrehajtásra kerül. Minden olyan objektumnak, ami a `Product` osztály példánya, van egy lokális `Logger` osztálybeli objektuma, amely a naplózás elvégzésére hivatott. Ezzel azonban az egységbezárás elve megsérült, mivel a `Product` osztály nem csak azt a követelmény teljesíti, amiért létrejött. Az osztályt így keresztezik a vonatkozások, melyeket szét kell választani. Ez azonban nem egy új igény, s éppen ezért jöttek létre az eljárások, majd az osztályok is. Az ún. *átmetsző vonatkozás*okat viszont nem lehet ilyen módon szétválasztani. A naplózás épp egy ilyen átmetsző vonatkozás. Elég csekély az esélye annak, hogy egy valós rendszer esetében kizárólag a termék árának változását kell naplózni. Lehetséges a `Product` osztályból egy másik osztály származtatása, adott esetben egy terméké, melynek még több attribútuma van, s ezek módosításainak naplózása is követelménnyé válik. Ennek következménye, hogy egyre több helyen jelenik meg a naplózáshoz tartozó kód, megnehezítve ezzel a későbbi módosítást, és a követelmények ily

módon történő keveredése áttekinthetetlen kódot eredményez [10].

Ilyen, gyakran előforduló átmetsző követelmények még a kivételkezelés, vagy a hitelesítés és engedélyezés (angolul: authentication and authorization), a validáció.

2.1.3 Az OO problémáinak megoldása – AOP

A fentebb bemutatott problémák mind szükségessé tették egy új programozási technika létrejöttét, mellyel lehetővé válik napjaink komplex, leginkább üzleti alkalmazásainak oly módon történő megvalósítása, mely alapjaiban az objektum-központú programfejlesztést használja, azonban mégsem kell áttörje az azáltal támasztott specifikációkat. Ezen szükséglet megvalósítása eredményezte az aspektusorientált programozás kialakulását, létrejöttét. Miután az OOP egy nagyszerű lépcsőfok a programozás fejlődése során, s bemutatott problémái leginkább hiányosságként tekinthetők, mintsem tervezési vagy használhatósági hibákként, az aspektusorientált programozás célja annak megfelelő kiegészítése, nem pedig teljes megreformálása. Az AOP segítségével tehát lehetővé vált az OOP megtartása, s alkalmazása csak azokon a pontokon válik szükségessé, ahol az OOP segítségével az alkalmazás csak a szabályok áthágásával lenne megvalósítható.

2.2 Az AOP általános jellemzői

Alkalmazás fejlesztése során az összes követelmény maradéktalan kielégítése a cél. Ezeket tehát muszáj megvalósítani ahhoz, hogy a készülő alkalmazás megfelelő legyen, de – mint azt már említettem – akadnak olyan követelmények is, melyek nem kapcsolódnak szorosan az alkalmazás valódi céljához, funkciójához, mégis elengedhetetlenek. Ezeket *nemfunkcionális követelmény*eknek hívjuk. Sajnos bármilyen nyelven is készüljön az adott program – legyen az objektumorientált, vagy sem –, a különböző típusú követelmények azonos módon történő megvalósítása bonyolult, kusza kódot eredményez [11].

Erre a problémára kínál megoldást az aspektusorientált programozás. Elsődleges célja a vonatkozások szétválasztása. Az ily módon történő fejlesztés különbözik a megszokottól. Egy alkalmazás általában egy olyan nyelv felhasználásával kerül megvalósításra, mely a legkevesebb erőforrás ráfordításával teszi azt lehetővé. Sok esetben a teljes fejlesztői csoport is egy adott nyelvre specializálódik. Az elsődleges követelmények megvalósítása ezért a választott nyelven történik, az átmetsző követelményeké pedig az aspektusorientált nyelven.

Nem lényeges elem az elsődleges követelmények megvalósítására választott nyelv, mindössze az fontos, hogy az átmetsző követelmények kódja összeilleszthető legyen az alkalmazásával, előállítva így a teljes rendszert. Bevett gyakorlat azonban, hogy maga az aspektusorientált rész is egy a fő alkalmazás során felhasznált nyelven íródik.

Mint minden nyelvnek, annak is, mely a vonatkozásokat valósítja meg, rendelkeznie kell *specifikáció*val és *implementáció*val.

2.2.1 Egy AOP rendszer specifikációja

Egy aspektusorientált rendszer fő komponensei:

- *Csatlakozási pont*ok (angolul: join points)
- Egy nyelv, melyben a pontok megtalálhatóak
- *Tanács* (angolul: advice), bár funkcióját tekintve teendőnek is nevezhetjük
- Egy komponens – pl. egy osztály –, mely megvalósítja a bezárást

2.2.1.1 Csatlakozási pontok

Ezek azok a pontok, ahol az alkalmazás és az *aspektus*ok találkoznak. Ez tehát egy pontosan meghatározott hely az alkalmazás futási folyamatában, ahol a keresztező vonatkozás egyébként megjelenne. Lehetnek ezek metódus- vagy konstruktor hívások, kivételkezelők, vagy bármely más azonosítható pont a program futása közben.

Olyan alkalmazás esetén, ahol adatbázisba kerülnek adatok, fontos követelmény lehet a naplózás. Tételezzük fel, hogy az adatbázist célzó tranzakciókat megvalósító osztályban található az `updateTable()` metódus, ami kezeli az összes táblafrissítést. A naplózási követelmény teljes megvalósítása érdekében itt kell elhelyezkedjen a naplózással kapcsolatos kód is. Ez lehet egy belépési pont, ami az osztály és a metódus nevével adható meg.

Egy ilyen lehetséges csatlakozási pont:

```
public String DBTrans.updateTables(String);
```

Ennek megadása a különböző nyelvek esetében más és más, mindössze az a fontos, hogy pontosan definiált legyen az a hely, ahol megjelenik az átmetsző vonatkozás.

2.2.1.2 Vágáspont

A *vágáspont*ok (angolul: pointcut) tartalmazzák és fogják egybe az alkalmazás belépési pontjait. Mivel a belépési pontok konkrétan az alkalmazás kódjában helyezkednek el, szükségessé válik egy olyan konstrukció, mely meghatározza az aspektusorientált nyelv számára, hogy mikor illessze a belépési pontokat. Ez tehát egy olyan kifejezés, ami az egyes programpontokat kiválasztja, és meghatározza azokat a programhelyeket, ahol az aspektust alkalmazni kell.

Ilyen AspectJ-ben definiált vágáspontok lehetnek például:

- `execution(* get*(*, *) )` - olyan metódusok végrehajtása, melyek neve "get"-tel kezdődik, és két paraméterrel rendelkeznek
- `within(com.company.*)` - minden belépési pont a com.company csomagon belül

2.2.1.3 Tanács / Teendő

A legtöbb AOP specifikáció esetében a tanács kódja három különböző helyen hajtható végre: a belépési pont illeszkedése előtt, helyett és után. A vágáspont illeszkedése váltja ki a tanács végrehajtását. Példaképp egy `before` tanács:

```
before(String s) : updateTables(s) {
     System.out.println("Passed parameter – " + s);
}
```

Amennyiben a vágáspont elő lett idézve, a tanács kódja végrehajtódik. A fenti példában a tanács a belépési pont kódjának lefutása előtt végrehajtódik. A `String` típusú argumentum átadásra kerül, tehát szükség esetén használható.

2.2.1.4 Aspektusok

Az aspektus (angolul: aspect) feladata, hogy egy egységbe zárja a vágáspontokat és a tanácsokat. Az aspektusokat az osztályokhoz hasonló módon kell létrehozni, és ezek is arra hivatottak, hogy az egy követelményhez tartozó kódokat egy egységbe zárják. Példa egy aspektusra:

```
public aspect TableAspect {
     pointcut updateTable(String s) :
```

```
        call(public String DBTrans.updateTables(String)
        && args(s));
        before(String s) : updateTable(s) {
             System.out.println("Passed parameter - " + s);
        }
    }
```

A `TableAspect` egy olyan aspektus, mely egy az `updateTables` metódushoz tartozó vonatkozást implementál. Minden funkcionalitás, mely ehhez a vonatkozáshoz kapcsolódik, egy helyen található, be van zárva egy saját struktúrába.

2.2.2 Egy AOP nyelv implementációja

A vonatkozások aspektusorientált nyelven történő megírása után a következő feladat az elsődleges alkalmazás és az aspektusok összedolgozása egy kész rendszerré. Ez a folyamat a *szövés* (angolul: weaving). Típusai – attól függően, hogy az mikor történik meg:

- Fordítási idejű
- Betöltési idejű
- Futási idejű

Az első esetben az elsődleges- és az aspektusorientált forráskód szövése a byte-kód létrejötte előtt megtörténik. A betöltési idejű szövés esetén az összefésülés csak az osztályok betöltésekor történik meg, azonban ez is a byte-kód szintjén. A futási időben történő szövésnél viszont a virtuális gép felel a belépési pontok felderítéséért, valamint a végrehajtandó kódok beillesztéséért.

Példaként véve egy Javában írt alkalmazást, az osztályok és a lehetséges aspektusok mind egy vonatkozást valósítanak meg. Maga az alkalmazás a "javac" fordító által kerül byte-kódra fordításra. Ezután a byte-kód már futtatható a JRE (Java Runtime Environment – Java futtatókörnyezet) által. Emellett azonban az aspektusoknak is futniuk kell. Mivel az aspektusok kódja is Java-ban íródott, nem elképzelhetetlen, hogy egy fordító átkonvertálja az aspektusok kódját tiszta Java kóddá, s maguk az aspektusok osztályokká, a belépési pontok, a vágáspontok és egyéb leírók pedig valamilyen szintén Java konstrukcióvá alakuljanak. Ha ez megtörtént, a szabványos Java fordító használható az aspektusok byte-kóddá történő átalakítására is [10].

Feltéve, hogy rendelkezésre áll egy olyan fordító, mely mind a Java és az aspektus-kódokat képes byte kóddá konvertálni, még mindig szükséges azok valamilyen módon történő összekapcsolása.

Fordítási időben történő összefésülés esetén a fordító megvizsgálja az aspektus kódját, amennyiben szükséges, átkonvertálja azt az elsődleges nyelvre, majd beilleszti az alkalmazás kódjába.

Az előző példánál maradva, ahol az `updateTables()` metódushoz egy csatlakozási pont lett definiálva, valamint egy vágáspont, hogy az még az `updateTables()` metódus lefutása előtt végrehajtódjon, a futási idejű összefésülő megtalálva az `updateTables()` metódust, a tanács kódját beilleszti a metódusba. Amennyiben az aspektus osztállyá lett konvertálva, a metódusban történő hívás lehet egy hivatkozás az új aspektus objektum egy metódusára.

Összefésülés után a következőképp nézhet ki a kód:

```
public String updateTables(String SQL) {
      //start code inserted for aspect
      TableAspect.getAspect().updateTable(SQL);
      //end code inserted for aspect
      initializeDB();
      sendSQL(SQL);
}
```

A fenti példában így egy a `TableAspectClass` osztály `updateTable` metódusára irányuló hívás kerül elhelyezésre, mely a korábban definiált `TableAspect` aspektus-kódból jött létre. Ezt a feladatot egy előfordító végzi el. Ha az aspektus kódja már bele lett szőve az alkalmazás kódjába, a létrejövő átmeneti fájlok kerülnek a Java fordítóhoz, s az így keletkező kód már megvalósítja mind az elsődleges, mind az átmetsző követelményeket.

Érdekességképp megjegyzem, hogy az AspectJ 2001-es megjelenésekor kizárólag forráskód-szintű szövővel rendelkezett. 2002-ben ez egy osztályonkénti byte-kód szövővel bővült, majd az AspectWerkz [12] 2005-ös integrációját követően már a fejlettebb, betöltési idejű szövés is lehetségessé vált. A futási idejű szövés azonban még mindig nem támogatott.

2.3 Az AOP technológiái és fejlesztői eszközei

Mint az az új technológiák esetében történni szokott, az AOP esetén is több keretrendszer

jött létre, melyek többnyire saját fejlesztői eszközökkel rendelkeznek. Ezek általában más-más funkcionalitásokat tartalmaznak, s többnyire más nyelv használatát kívánják meg a fejlesztőtől mind az alkalmazás, mind az aspektusok elkészítése során. Miután az előzőekben általánosan bemutattam az AOP kialakulását és jellemzőit, az egyik ilyen keretrendszert szeretném részletesebben is jellemezni.

2.3.1 A Coldspring keretrendszer

A következőkben a ColdSpring Framework [13] keretrendszert kívánom bemutatni, mely a ColdFusion (CF) [14] alkalmazásfejlesztő platform kiegészítője. Habár nem a legelterjedtebb, választásom mégis erre esett, mivel az általam készített alkalmazás fejlesztése során ezt használtam fel.

Segítségével egyszerűbbé válik a *ColdFusion komponens*ek (angolul: CFC – ColdFusion Component) függőségeinek kezelése, s egyben az első CF-hez kapcsolható AOP keretrendszer is. Legfőbb funkciói a függőség-beágyazás, illetve az AOP kiegészítés. A következőkben ezen két funkciót kívánom részletesen bemutatni. Természetesen a keretrendszer számos más lehetőséget is tartogat, ám ezek részletes ismertetése leginkább egy könyv terjedelmét igényelné. A funkciók részletes leírása megtalálható a ColdSpring hivatalos dokumentációjában [15].

Fontosnak tartom megjegyezni, hogy miután a ColdSpring alapvetően hasonlít a Javához használható Spring keretrendszerre, az elnevezéseket is sok esetben onnan kölcsönözte a konzisztencia megőrzésének érdekében. Így a már Javában is ismert *bab* fogalma itt is megjelenik, s a későbbiekben gyakran használni is fogom azt. ColdSpring esetében a bab alatt olyan ColdFusion komponenst értünk, mely rendelkezik a tulajdonságainak lekérdezéséért (angolul: getter) és beállításáért (angolul: setter) felelős metódusokkal.

2.3.1.1 Függőség-beágyazás

Nagyobb alkalmazások készítése esetén szinte elkerülhetetlenül előáll az a helyzet, hogy számos komponenst kell létrehoznunk, s a létrehozott komponensek között – sokszor többszörös – függőségi kapcsolat jön létre.

A következőkben bemutatom ezen függőségek kialakulását, s a függőségek kezelésének lehetőségeit egy egyszerű példán keresztül. Ehhez néhány ColdFusion komponenst fogok

definiálni.

SimpleService.cfc komponens:

```
<cfcomponent name="Simple Service" hint="This is a Simple
Service, which is how external things interact with the
Model.">
     <cffunction name="init" access="public"
     returntype="any" hint="Constructor.">
          <cfreturn this />
     </cffunction>

     <cffunction name="getSimpleGateway" access="public"
     returntype="any" output="false" hint="This will
     return the SimpleGateway.">
          <cfreturn variables.instance['simpleGateway'] />
     </cffunction>

     <cffunction name="setSimpleGateway" access="public"
     returntype="void" output="false" hint="This will set
     the SimpleGateway.">
     <cfargument name="simpleGateway" type="any"
     required="true" hint="SimpleGateway" />
          <cfset variables.instance['simpleGateway'] =
          arguments.simpleGateway />
     </cffunction>
</cfcomponent>
```

SimpleGateway.cfc komponens:

```
<cfcomponent name="Simple Gateway" hint="This is a  Simple
Gateway to interact with a database.">

     <cffunction name="init" access="public"
     returntype="any" hint="Constructor.">
          <cfreturn this />
     </cffunction>

     <cffunction name="getConfigBean" access="public"
     returntype="any" output="false" hint="This will
     return the ConfigBean.">
          <cfreturn variables.instance['configBean'] />
     </cffunction>

     <cffunction name="setConfigBean" access="public"
     returntype="void" output="false" hint="This will set
     the ConfigBean.">
     <cfargument name="configBean" type="any"
```

```
        required="true" hint="ConfigBean" />
            <cfset variables.instance['configBean'] =
            arguments.configBean />
        </cffunction>
    </cfcomponent>
```

ConfigBean.cfc komponens:

```
    <cfcomponent name="Config Bean" hint="Contains application
    configuration data, such as datasource name.">

        <cffunction name="init" access="public"
        returntype="any" hint="Constructor.">
        <cfargument name="datasourceName" type="string"
        required="false" default="MyDSN" />
            <cfset
            setDatasourceName(arguments.datasourceName) />
            <cfreturn this />
        </cffunction>

        <cffunction name="getDatasourceName" access="public"
        returntype="string" output="false" hint="This will
        return the DatasourceName.">
            <cfreturn
            variables.instance['datasourceName'] />
        </cffunction>

        <cffunction name="setDatasourceName" access="public"
        returntype="void" output="false" hint="This will set
        the DatasourceName.">
        <cfargument name="datasourceName" type="string"
        required="true" hint="DatasourceName" />
            <cfset variables.instance['datasourceName'] =
            arguments.datasourceName />
        </cffunction>
    </cfcomponent>
```

Amint az a fent deklarált komponensek kódjából kitűnik, az egyes komponensek között egyfajta függőség alakul ki. A `SimpleService` komponensnek szüksége van a `SimpleGateway` komponensre, míg a `SimpleGateway` komponens a `ConfigBean` komponens meglétét igényli. Lehetséges ugyanis, hogy a `SimpleService` a `SimpleGateway` komponenst kell meghívja egy adatbázis-művelet elvégzése érdekében, míg a `SimpeGateway` komponens a `ConfigBean`-t fogja meghívni, hogy megtudja az adatbázis eléréséhez szükséges `ColdFusion` adatforrás nevét.

Alapesetben a rendszer működéséhez nekünk kell konfigurálni ezeket a komponenseket, hogy a függőségek feloldódjanak. Ehhez a következő CF utasítások végrehajtására lenne szükség:

```
<cfscript>
     configBean = CreateObject('component',
     'ConfigBean').init();
     simpleGateway = CreateObject('component',
     'SimpleGateway').init();
     simpleService = CreateObject('component',
     'SimpleService').init();
     simpleGateway.setConfigBean(configBean);
     simpleService.setSimpleGateway(simpleGateway);
</cfscript>
```

Természetesen a fenti parancsokkal létrehozott objektumok példányosítási sorrendje nem mindegy. Emellett nekünk kell átadni az egyes függőségeket a megfelelő CFC-knek. Ugyan a bemutatott példában mindez néhány sornyi kód megírását jelentette, nagyobb alkalmazások esetén ez többszörösére növekedhet. Ilyen esetben a szükséges kód manuális elkészítése szinte lehetetlen feladat. Ilyenkor jól jön a ColdSpring nyújtotta *függőség-beágyazás*. Ahelyett, hogy nekünk kellene kezelni a függőségeket, mindössze annyit kell tennünk, hogy „megmondjuk” a ColdSpring-nek, hogy az egyes komponenseinknek milyen függőségei vannak, s a munka nehezétől így meg is szabadulunk. Ehhez egy XML fájlt kell létrehozzunk, amely a fenti példának megfelelően lehet a következő ColdSpring.xml:

```
<beans default-autowire="byName">
     <bean id="simpleService" class="SimpleService" />
     <bean id="simpleGateway" class="SimpleGateway" />
     <bean id="configBean" class="ConfigBean" />
</beans>
```

A példában látható, hogy mindössze a komponenseink elérési útját kell megadnunk (amik jelen esetben maguk a komponensek nevei, feltételezve, hogy a komponensek a webkiszolgáló gyökérkönyvtárában (angolul: webroot) helyezkednek el), valamint ezekhez egy-egy azonosítót (`"bean id"`) rendelünk. A ColdSpring a `default-autowire="byName"` attribútumnak köszönhetően át fogja vizsgálni a komponenseinket, megkeresi azok beállításért felelős függvényeit (angolul: setter), illetve azokat a komponenseket, melyek `"bean id"`-je megegyezik ezen függvények nevével. Például a `setSimpleGateway()` függvényt megtalálva olyan babot fog keresni, melynek `"bean`

id"-je „simpleGateway”. Ez a bab pedig a SimplGateway komponenst tartalmazza. Találat esetén a ColdSpring automatikusan elvégzi a beállítást. Ezt a funkciót nevezzük név szerinti függőség-beágyazásnak.

Ahhoz, hogy a ColdSpring elvégezze a feloldást, még szükséges a keretrendszer inicializálása, a ColdSpring factory komponens példányosítása, valamint a konfiguráció betöltése. Ezt a következő CF kóddal tehetjük meg:

```
<cfset coldSpringConfig =     '/ColdSpring.xml' />
<cfset beanFactory = CreateObject('component',
'coldspring.beans.DefaultXmlBeanFactory').init() />
<cfset beanFactory.loadBeans(coldSpringConfig) />
```

A fenti utasítások lefutása után már használhatóak a komponensek, s nincs szükségünk a függőségek kézi feloldására. A létrejövő beanFactory objektum tartalmazni fogja a korábban az XML fájlban deklarált babokat. A következő kódban látható, hogyan használható az így létrejött ColdSpring factory példány:

```
<!--- Get the Simple Service CFC from ColdSpring. --->
<cfset simpleService =
beanFactory.getBean('simpleService') />

<!--- Get the Gateway that ColdSpring already has set into
the Simple Service. --->
<cfset simpleGateway = simpleService.getSimpleGateway() />

<!--- And get the Config Bean that ColdSpring has set into
Simple Gateway. --->
<cfset configBean = simpleGateway.getConfigBean() />
```

Habár az itt bemutatott példa során nem feltétlenül mondható el, hogy az ily módon megírt kód lényegesen rövidebb lenne, természetesen egy terjedelmesebb alkalmazás esetében nagyságrendbeli különbségek adódhatnak.

Nem szabad azonban elfelejteni azt sem, hogy ezen lehetőség is rendelkezik hátrányokkal. Használata mindenképp körültekintést igényel. Néhány eset, amikor ezen funkció használata nem lehetséges:

- Nem minden esetben jó ötlet, ha hagyjuk a ColdSpringet, hogy nevek egyezése alapján „összedrótozza” minden komponensünket. Előfordulhat, hogy egyik komponensünk rendelkezik egy setConfig() metódussal (melynek semmilyen

kapcsolata nincs más komponensekkel), s emellett a konfigurációs állományunkban létrehoztunk „config” nevű babot. Nem ritka eset az sem, hogy több komponensünk is tartalmaz azonos nevű beállításért felelős függvényt. Ilyen esetekben a ColdSpring ugyan elvégzi a függőség-beágyazást, azonban az eredmény egyáltalán nem az elvártak szerinti lesz.

- A beállításért felelős függvények nevei nem feltétlenül egyeznek a babok azonosítóival. Ilyenkor a ColdSpring természetesen nem fogja tudni elvégezni a feladatot.
- Sok fejlesztő szereti explicit módon elvégezni a függőségek megadását. Ilyen esetekben a ColdSpring ugyanúgy el fogja végezni a függőségek kezelését, mindössze azok kézi megadása szükséges. Ezzel azonban a több munkáért cserébe egy olyan XML fájlhoz jutnak, mely mintegy térkép, reprezentálja az alkalmazás felépítését, az egyes komponenseket, s azok egymástól való függését.

A fenti esetek kapcsán azonban elmondható az is, hogy a ColdSpring rendelkezik olyan lehetőséggel, mely során a függőségeket saját magunk, explicit módon definiáljuk, s azok a keretrendszer által feloldásra kerülnek.

2.3.1.2 AOP

A ColdSpring másik nagyon fontos, és talán leghasznosabb része az AOP kiegészítés. Segítségével megvalósíthatóak a már korábban említett átmetsző követelmények. Ezen tulajdonságot nem kívánom nagy részletességgel ismertetni, mivel az általam készített alkalmazás fontos elemét képzi, s annak későbbi bemutatása során ezt megteszem. A leggyakoribb ilyen követelmény a naplózás, így tekintsük például a következő egyszerű StringService.cfc komponensben elhelyezkedő függvényt:

```
<cffunction name="toCapital" access="public"
returntype="string" output="false">
<cfargument name="string" type="string" required="true" />

     <!--- Log method and arguments. --->
     <cfset getLogger().log('toCapital', arguments) />

     <cfreturn Ucase(arguments.string) />
</cffunction>
```

Amint az a kódrészletből kiderül, a függvény az argumentumként megkapott String-et adja vissza nagybetűssé alakítva. A visszatérési pont előtt meghívásra kerül egy másik komponens függvénye, mely a naplózást végzi. Jelen példában ennek megvalósítása elég egyszerű volt, de gondoljunk bele, mennyi munkával járna ugyanez egy több száz vagy ezer függvényt tartalmazó alkalmazás esetén. Nem lenne elég ugyanis még a naplózást végző kódrész másolása-beillesztése sem, mivel minden esetben meg kellene változtatnunk a naplóbejegyzés forrásaként az aktuális függvény nevét jelölő argumentumot. Elképzelhető az is, hogy a későbbiekben változik az alkalmazásunk, s nem elég már kizárólag a meghívott függvény nevének, s argumentumainak naplózása, de szükségessé válik például a visszatérési értékek naplózása is.

Az AOP használatával azonban lehetővé válik az ilyen szükségletek nagyságrendekkel egyszerűbb módon történő megvalósítása. Először is szükséges egy olyan komponens elkészítése, mely a naplózási folyamat elvégzéséhez szükséges kódot tartalmazza. Legyen ez a komponens a LoggingAdvice.cfc:

```
<cfcomponent output="false" displayname="LoggingAdvice"
hint="Advise service layer methods and apply logging."
extends="coldspring.aop.MethodInterceptor">

     <cffunction name="init" returntype="any"
     output="false" access="public" hint="Constructor">
          <cfreturn this />
     </cffunction>

     <cffunction name="invokeMethod" returntype="any"
     access="public" output="false" hint="">
     <cfargument name="methodInvocation"
     type="coldspring.aop.MethodInvocation"
     required="true" />
          <cfset var local =  StructNew() />
          <cfset local.logData = StructNew() />
          <cfset local.logData.arguments =
StructCopy(arguments.methodInvocation.getArguments()) />
          <cfset local.logData.method =
arguments.methodInvocation.getMethod().getMethodName() />
          <cfset request.logData = local.logData />

          <!--- Call the original method --->
          <cfset local.result =
          arguments.methodInvocation.proceed() />
```

```
            <cfreturn local.result />
        </cffunction>
    </cfcomponent>
```

A komponens felépítése egyszerű, főként ha azt is figyelembe vesszük, hogy ezzel már egy akármilyen méretű alkalmazás esetén megvalósítható a kívánt funkció. Mindössze annyi szükséges, hogy „megmondjuk” a ColdSpring-nek, hogy az általunk kijelölt komponens(ek) minden – vagy csak bizonyos – függvényének hívása esetén szeretnénk, ha ez a `LoggingAdvice` komponens is futtatásra kerülne. Ezt követően a ColdSpring létrehoz egy „AOP átjárót” (angolul: AOP Proxy), s mintegy átveri az alkalmazás többi részét. Ezt követően ugyanis az adott komponensre irányuló összes kérés ezt az AOP Proxy-t fogja visszakapni. A hívást végző komponensek erről semmit sem fognak tudni, s számukra az AOP Proxy az eredeti komponens.

A fentebb bemutatott `LoggingAdvice` komponens mindössze eltárolja a számára szükséges információkat, ezt követően továbbítja a hívást az eredeti komponens felé, majd végül megkapva attól a visszatérési értéket, ő is visszaadja azt. Itt tehát lehetőségünk nyílik arra, hogy pontosan irányítsuk, mi történik a hívás előtt, helyett és után.

A funkció működéséhez ez esetben is szükséges egy konfigurációs XML fájl elkészítése, mely legyen a következő ColdSpring.xml:

```
    <beans>
        <bean id="stringService"
        class="coldspring.aop.framework.ProxyFactoryBean">
            <property name="target">
                <bean class="StringService" />
            </property>
            <property name="interceptorNames">
                <list>
                    <value>loggingAdvisor</value>
                </list>
            </property>
        </bean>

        <bean id="loggingAdvice" class="LoggingAdvice" />

        <bean id="loggingAdvisor"
class="coldspring.aop.support.NamedMethodPointcutAdvisor">
            <property name="advice">
                <ref bean="loggingAdvice" />
```

```
            </property>
            <property name="mappedNames">
                 <value>*</value>
            </property>
        </bean>
    </beans>
```

A kódban definiálunk egy `loggingAdvice` azonosítójú babot, mely a `LoggingAdvice` komponenst reprezentálja. A `loggingAdvisor` bab feladata, hogy az `advice` attribútumnál beállított `loggingAdvice` babot tanácsként hozzárendelje a `mappedNames` tulajdonságánál beállított függvényekhez (ami jelen esetben az összes függvényt jelöli). A `stringService` bab végzi el az AOP Proxy létrehozását, hozzárendelve a `loggingAdvisor` babot a `StringService` komponenshez. Ezt követően a ColdSpring factory példányosítása és az új bab lekérése után használhatóvá válik a Proxy:

```
<cfset coldSpringConfig =     '/ColdSpring.xml' />
<cfset beanFactory = CreateObject('component',
'coldspring.beans.DefaultXmlBeanFactory').init() />
<cfset beanFactory.loadBeans(coldSpringConfig) />

<cfset stringService =
beanFactory.getBean('stringService') />

Capitalized version of heLLo is:
#stringService.toCapital('heLLo')#
```

Mint az a fenti kódból látható, a komponens ily módon létrejövő AOP Proxy-n keresztüli használata nem különbözik az egyébként megszokottól, kódunk funkcionalitása azonban nagymértékben megnőtt, így ugyanis már a `LoggingAdvice` komponensen belül elhelyezett kódok is lefutnak.

2.4 Az AOP hátrányai

Mint az minden új technológia esetében megfigyelhető, a fejlesztők szeretnék azt azonnal alkalmazni, sok esetben a már meglévő kódjukat is ezekkel kiegészíteni, átformálni. Természetesen az AOP esete sem kivétel ez alól. Ez sem használható minden egyes probléma(kör) megoldására, ezt pedig mindenképp figyelembe kell venni, mielőtt alkalmazni szeretnénk. Ha egy feladatot nehéznek tartunk megvalósítani az AOP felhasználásával, elképzelhető, hogy nem is igazán annak alkalmazására van szükség. Amit egyes emberek

ugyanis hátránynak tekintenek, azt mások gyakran nem megfelelő használatnak neveznek. Nem tekinthetjük ugyanis egy kalapács hátrányának, hogy nem használható csavarhúzóként.

2.4.1 Gyakorlati alkalmazhatóság

Amint azt már korábban is említettem, szükségesek bizonyos folyamatok ahhoz, hogy egy adott alkalmazás az aspektusokkal történő kiegészítést követően az elvárt módon, kiegészített funkcionalitással együtt futtatható legyen. Hogyan rendelhető egy metódushoz egy aspektust annak megváltoztatása nélkül? Néhány lehetséges megoldás:

- Automatikus kód generálás (a legkönnyebb lehetőség)
- A lefordított tárgy- vagy byte-kód manipulálása
- Dinamikus futásidejű proxy-k használata (nem minden nyelvben érhető el)

Ugyan sok AOP keretrendszer elérhető, elmondható, hogy többségük Java-specifikus. Megfigyelhető az is, hogy ilyen keretrendszerek leginkább olyan nyelvekhez léteznek, melyek jó futásidejű introspektív támogatással rendelkeznek. Így tehát ha valaki ilyen technológiát szeretne alkalmazni, rá van kényszerítve, hogy csupán néhány nyelv közül választhat.

2.4.2 Nyelvi elemek

Mivel az aspektusorientált programozás lényeges eleme, hogy az alkalmazás futása során megadhatunk olyan pontokat, melyekhez különböző aspektusokat rendelünk, gondot okozhatnak olyan helyzetek, amikor nem tehetjük meg ezen pontok kijelölését. Léteznek ugyanis olyan nyelvi elemek, melyek esetében a csatlakozási pont definiálása nem lehetséges.

Tekintsünk példaként egy olyan alkalmazást, melyben egyszerre több párhuzamos feldolgozási folyamatnak kell futnia. Ilyen esetben az alkalmazásbon *szál*akat (angolul: thread) hozunk létre, s így szükségessé válhat a szinkronizáció. Ezt elérhetjük a folyamaton belüli kommunikációval (angolul: IPC – Inter-Process Communication) [16], vagy más nyelvi elemek felhasználásával is. Így megszüntethetővé válnak a versenyhelyzetek, azonban előállhatnak olyan helyzetek, mikor az alkalmazásunk megáll, mivel a folyamatok kölcsönösen egymásra várnak. Ezt az állapotot nevezzük *holtpont*nak (angolul: deadlock) [17]. Számos technika létezik ezen állapotok felismerésére és elkerülésére.

Java-ban az ilyen szinkronizáció megvalósításának legelterjedtebb módja, hogy minden

egyes kritikus kódrész előtt elhelyezzük a `synchronized` kulcsszót. Ezáltal a futás során az objektum adott példánya zárolásra kerül, s semelyik másik szál nem futtathat kódot a többi kritikus részben (melyek ugyanazzal a kulcsszóval lettek ellátva). Sajnos az ilyen módon definiált konstrukció esetében nem tudunk csatlakozási pontokat kijelölni.

2.4.3 Tanulási folyamat

Mivel az aspektusorientált programozás egy teljesen új technológia, érthető hogy használata a kezdeti stádiumban bonyolult. Emellett az AOP egy módját mutatja meg annak, ahogy a szoftverfejlesztési folyamatra gondolhatunk. Ezt pedig a klasszikus OO programozáshoz szokott fejlesztőknek sok esetben nem könnyű elsajátítani. Így tehát szükségessé válhat egy hosszadalmasabb tanulási folyamat is, mire az új gondolkodásmódot, eszközhasználatot elsajátítanánk. Nem vitás azonban, hogy ez hosszú távon kifizetődő.

3. Az alkalmazás felépítése

Az általam elkészített alkalmazás egy zárt levelező rendszer, mely a napjainkban egyre inkább elterjedt webalkalmazás formájában került megvalósításra, s formájából eredően két fő komponensből tevődik össze:

- Kliens-oldali alkalmazás
- Szerver-oldali alkalmazás és erőforrások

Az elkészített szoftver amellett, hogy önmagában is használható, akár egy vállalatirányítási rendszernek is szerves részévé válhat, melyek napjainkban egyre nagyobb népszerűségnek örvendenek, s felhasználásuk többnyire elengedhetetlen egy szervezet működése során.

Ilyen alkalmazások esetében a szétválasztást nem csak az erőforrások megosztása, a funkciók szétválasztása, de a biztonság növelése is indokolttá teszi. Általában egy alkalmazás teljesíti az összes kívánt követelményt, azonban az ilyen üzleti-, illetve webalkalmazások esetén a funkciók nagy többsége a szerver oldalon helyezkedik el. A kliens feladata többnyire a megjelenítés, és a felhasználóval történő interakció. Az üzleti logikát, az alkalmazás folyamatait pedig a szerveren elhelyezkedő komponensek valósítják meg. Mivel a felhasználó oldalán semmilyen fontos, az alkalmazás fő funkcióit megvalósító feladat megvalósítása sem történik, így az illetéktelenek rendszerre irányuló behatolási, károkozási kísérletei is nehezebbé válnak. Ennek oka, hogy ily módon a komponensek többsége láthatatlan marad a felhasználó számára, s az alkalmazás esetleges visszafejtésével sem lehet megtudni szinte semmilyen fontos információt annak működéséről, felépítéséről.

A szoftver megvalósítása során a következő eszközök kerültek felhasználásra:

- Adobe Flex Builder 3
- Adobe AIR SDK
- Adobe ColdFusion 9
- ColdSpring Framework 1.2 stable
- Apache2.2.11 PHP/5.2.6-3ubuntu4.2 with Suhosin-Patch JRun/4.0 Webserver

- MySQL 5.0.75
- Ubuntu 9.0.4 Linux server, Kernel: 2.6.28-11-server

A fejlesztés folyamata négy fő lépésre osztható:

1. Az alkalmazás megtervezése, a kiszolgáló adatbázis létrehozása
2. A Szerver oldali alkalmazás fejlesztése
3. A Kliens oldali alkalmazás létrehozása
4. Az aspektusorientált rész elkészítése a szerver-alkalmazáshoz

3.1 Szerver-oldali alkalmazás

A szerver oldali alkalmazás tartalmazza majdnem az összes funkcionalitást, mellyel az alkalmazás rendelkezik. A tervezés és fejlesztés során így ennek kell a legnagyobb figyelmet szentelni. Első lépésben – az alkalmazás elsődleges funkcióit alapul véve – az adatbázist terveztem meg, majd hoztam létre. Ezt követően készültek el a szerver-alkalmazás komponensei. Az alkalmazás főbb funkció a következők:

- Felhasználókezelés / Autentikáció
- Üzenetek kezelése (küldés / fogadás)

3.1.1 Tervezés és az adatbázis létrehozása

A tervezés során a megvalósítani kívánt funkciókat figyelembe véve négy tábla elkészítését tartottam szükségesnek:

1. „users” – A felhasználók adatainak tárolásához
2. „letters” – A levelek tárolásához
3. „switch” – A levelek felhasználókhoz való rendeléséhez
4. „log” – A naplóbejegyzések tárolásához

Az adatbázis tárolásához a MySQL [18] adatbázis-kezelő rendszert használtam fel, mivel az ingyenesen elérhető, s létezik az általam használt szerveren üzemelő Linux operációs rendszer alatt futtatható változata is.

Az elkészített táblák kezelése során nem tartottam szükségesnek tranzakciókezelés alkalmazását, ezért az összes elkészített tábla kezelését a MyISAM motor [19] végzi.

Az adatbázisok részletes bemutatása során az attribútumok oszlopban a következő rövidítéseket kívánom használni a létrehozásuk során használt SQL utasítások jelölésére:

- NN – `NOT NULL` (az adott mező értéke nem lehet NULL)
- AI – `AUTO_INCREMENT` (a mező értéke automatikusan növekszik új sor hozzáadása esetén)
- D_0 – `DEFAULT '0'` (a mező alapértelmezett értéke '0')
- D_N – `DEFAULT NULL` (a mező alapértelmezett értéke NULL)
- D_CT – `DEFAULT CURRENT_TIMESTAMP` (a mező alapértelmezett értéke az aktuális dátum-idő)
- PK – `PRIMARY KEY` (az oszlop a tábla elsődleges kulcsa)

3.1.1.1 A „users” tábla

Mivel a legfontosabb funkció a felhasználók kezelése, illetve az ezzel együtt járó hitelesítés, az első lépés a felhasználók adatait tartalmazó tábla elkészítése volt.

A „users” tábla felépítése a következő:

Mező neve	Típusa	Attribútumok	Funkciója
userName	VARCHAR(20)	NN	Felhasználónév
userId	INT(10)	NN, AI, PK	Felhasználó azonosító (egyedi)
password	VARCHAR(20)	NN	Jelszó
adminRights	BOOL	NN, D_0	Adminisztrációs jogokkal rendelkezik?
regTime	TIMESTAMP	NN, D_CT	Regisztráció időpontja
lastLogin	TIMESTAMP	NN	Utolsó belépés időpontja
realName	VARCHAR(30)	NN	Felhasználó valódi neve

1. Táblázat: A „users” tábla szerkezete

Az 1. táblázatot megvizsgálva látható, hogy a felhasználó azonosító automatikusan növekszik minden új bejegyzés hozzáadásakor, illetve hogy ez az oszlop egyben a tábla elsődleges kulcsa is. A regisztráció időpontja alapértelmezett értéke a `CURRENT_TIMESTAMP`, mely az aktuális dátum-idő értékre állítja az új bejegyzéseket. Ezzel az alkalmazásban nem szükséges külön kód írása a regisztráció implementálásánál.

3.1.1.2 A „letters” és a „switch” tábla

Az üzenetek kezeléséhez szükséges táblát két részben valósítottam meg, mivel ily módon kevesebb hely szükséges az üzenetek tárolásához, illetve egyszerűbbé válik az üzenetek kezelése is. Miután a küldő több címzettnek is küldhet egy levelet, érdemesebb a címzetteket egy külön táblában tárolni, ahol az egyes bejegyzések egy adott levélre vonatkoznak, mintsem több alkalommal tárolni ugyanazt az üzenetet. Ezáltal egyszerűen megoldható az olvasás idejét jelző tulajdonság kezelése is.

A „letters” tábla felépítése a következő:

Mező neve	Típusa	Attribútumok	Funkciója
letterId	INT(20)	NN, AI, PK	Levél azonosítója (egyedi)
letterText	TEXT	NN	Levél szövege
sender	VARCHAR(20)	NN	Küldő felhasználóneve
subject	TEXT		Levél tárgya
senderRemoved	BOOL	NN, D_0	A feladó törölte?

2. Táblázat: A „letters” tábla szerkezete

A 2. táblázatot megvizsgálva látható, hogy a levél azonosító automatikusan növekszik minden új bejegyzés hozzáadásakor, illetve hogy ez az oszlop egyben a tábla elsődleges kulcsa is.

A „switch” tábla kiegészíti a „letters” táblát. Funkcióját tekintve mintegy kapcsolóként szolgál, mely jelzi, hogy mely levelet kinek kell kézbesíteni, s ennek segítségével válik lehetővé az egy levél több címzetthez történő küldése esetén is az egy adott levélhez tartozó, címzettenként mégis különböző attribútumok beállítása.

A „switch” tábla felépítése a következő:

Mező neve	Típusa	Attribútumok	Funkciója
row_id	INT(11)	NN, AI, PK	Sor azonosító (egyedi)
letterId	INT(11)	NN	Levél azonosító
recipientRealName	VARCHAR(30)	NN	Címzett valódi neve
received	TIMESTAMP	D_N	A levél kézbesítésének és küldésének időpontja
read	TIMESTAMP		Az elolvasás időpontja

3. Táblázat: A „switch” tábla felépítése

A 3. táblázatot áttekintve látható, hogy a sor azonosító minden bejegyzésnél automatikusan növekszik, s egyben a tábla elsődleges kulcsa is. A levél azonosítóra ebben a táblában az egyetlen kikötés, hogy nem lehet NULL érték. Ennek oka, hogy az azonosítónak mindenképp a „letter” táblában található egyik levél azonosítójával kell megegyezzen. Az azonosító itt nem egyedi, mivel több sor is hivatkozhat egy adott levélre (pl.: körlevél vagy több címzett).

3.1.1.3 A „log” tábla

A „log” tábla létrehozása különbözik a többi táblától, mivel ennek elkészítésére a fejlesztés utolsó fázisában került sor, mivel erre már az aspektusorientált kód megvalósítása során volt szükség. A tábla a különböző szerver oldali hívások esetén megtörténő naplóbejegyzéseket tartalmazza.

A „log” tábla felépítése a következő:

Mező neve	Típusa	Attribútumok	Funkciója
details	TEXT	NN	Az esemény leírása
source	TEXT		Az esemény forrása / keletkezési helye
occured	TIMESTAMP	NN, D_CT	Bekövetkezésének időpontja
id	INT(11)	NN, AI, PK	Az esemény azonosítója

4. Táblázat: A „log” tábla felépítése

A kódot áttekintve látható, hogy a tábla automatikusan növekvő értéket tartalmazó oszlopa az azonosító, mely az elsődleges kulcs is. A keletkezési időpontot jelző oszlop alapértelmezett értéke a `CURRENT_TIMESTAMP`, mivel a naplózás közvetlenül az esemény keletkezésekor meg kell történjen, s ezáltal a szerver-alkalmazás komponenseinek fejlesztésekor az időpont beállításának létrehozása már nem szükséges.

3.1.2 ColdFusion komponensek

Miután az alkalmazás ColdFusion felhasználásával készült, a kód CFML (ColdFusion Markup Language) nyelven íródott. Ez egy ún. szkript nyelv (angolul: scripting language), ezáltal a kód magas szintű, könnyen áttekinthető, jól tagolt. Az alkalmazás ColdFusion komponensekből (CFC-k) épül fel. Két ilyen komponenst hoztam létre:

1. letters.cfc – A levelek kezeléséhez
2. users.cfc – A felhasználók kezeléséhez

Emellett elkészült két ColdFusion "template" (CFM) fájl is:

1. Application.cfm – A ColdSpring keretrendszer inicializálásához, valamint az alkalmazás során szükséges globális feladatok elvégzéséhez.
2. index.cfm – Az Application.cfm vezérlésének lehetővé tételéhez (a ColdSpring Framework és változók újrainicializálása), valamint az AOP rész működésének egyszerű demonstrációjához.

A szerver oldali alkalmazás felelős az adatbázis lekérdezéseinek, módosításainak elvégzéséért, az ún. üzleti logika (angolul: business logic) megvalósításáért, valamint a kliens-alkalmazás kiszolgálásáért. Az aspektusorientált kiegészítés szintén itt helyezkedik el, habár

annak megvalósítása ezen a kódon láthatatlan.

A következőkben be kívánom mutatni az itt létrehozott két komponens függvényeit. Mivel az összes függvény rendelkezik néhány azonos jelzővel, ezeket itt kívánom részletezni, s a továbbiakban nem térek ki azok ismételt magyarázatára:

- **remote**: A függvény távoli hívások kezelésére használható. Ez minden függvény esetében beállításra került, mivel mindegyiket kizárólag a távoli kliens-alkalmazás használja.
- **required**: A paraméterek esetében jelzi, hogy azok megadása kötelező-e. Majdnem minden esetben felhasználásra került, mivel a függvények nem rendelkeznek opcionális argumentumokkal.
- **Output**: `enabled` értéke jelzi, hogy a függvény futása során a benne található nem ColdFusion utasítások egyszerű kimenetként tekintendőek. `supressed` érték esetén ezek a részek nem kerülnek feldolgozásra.

Emellett a függvények SQL lekérdezéseket tartalmazó részeiben megtalálható egy `datasource="#APPLICATION.DSN#"` kódrészlet is, mely szintén közös ezen részekben. Az „`APPLICATION.DSN`” egy `DSN` nevű globális változó, melynek láthatósága (angolul: scope) az APPLICATION, vagyis a teljes alkalmazás. Feladata, hogy egyetlen helyről (az értékadás az Application.cfm-ben) legyen reflektálható az esetleges ColdFusion szerverben történő névváltoztatás. Értéke az alkalmazásban `dsMessenger`, mely a ColdFusion (CF) adminisztrációs felületén létrehozott `DataSource` (adatforrás) neve, ami a felhasznált „dbMessenger” nevű MySQL adatbázisra mutat.

3.1.2.1 A „users” komponens

A komponens feladata a felhasználókezelés, az adatbázis „users” táblájának kezelése (lekérdezések és módosítások). Kizárólag függvényeket tartalmaz. A következőkben bemutatom a „users.cfc” komponens függvényeit:

addUser

```
remote boolean addUser ( required string userName, required
string realName, required string password, required boolean
adminRights )
```

Output: enabled
Parameters:
userName: string, required, Felhasználónév
realName: string, required, Valódi név
password: string, required, Jelszó
adminRights: boolean, required, Adminisztrátori jogok

Az `addUser` függvény segítségével lehetséges új felhasználó létrehozása. Paraméterként a létrehozni kívánt felhasználó felhasználónevét, valódi nevét, jelszavát, illetve jogosultságát kell megadni. A visszatérési érték `Boolean` típus, mely igaz értékkel jelzi a felhasználó sikeres létrehozását.

checkLog

```
remote boolean checkLog ( required string acc, required string
pass )
```

Output: enabled
Parameters:
acc: string, required, Felhasználónév
pass: string, required, Jelszó

A `checkLog` függvény feladata a belépéskor történő autentikáció. Argumentumként a belépni kívánó felhasználó felhasználónevét, valamint jelszavát kapja meg. Visszatérési értéke `Boolean` típusú, mely igaz értékkel jelzi, ha a felhasználó hitelesítése sikerrel zárult.

deleteUser

```
remote void deleteUser ( required string acc )
```

Output: suppressed
Parameters:
acc: string, required, Felhasználónév

A `deleteUser` függvény végzi el egy adott felhasználó rendszerből való törlését. Egyetlen argumentuma a törlendő felhasználó felhasználónevét jelzi. Visszatérési értékkel nem rendelkezik.

getrealName

```
remote string getrealName ( required string userName )
```

Output: enabled
Parameters:
userName: string, required, Felhasználónév

A `getRealName` függvény segítségével lekérdezhető egy adott felhasználó valódi neve a felhasználónév alapján. Argumentuma a keresett felhasználónév. Visszatérési értéke `String` típusú, a paraméterként megadott felhasználónévhez tartozó valódi nevet tartalmazza.

getUsers

```
remote query getUsers ( )
```

Output: enabled

A `getUsers` függvény egy egyszerű lekérdezést valósít meg. Paraméterrel nem rendelkezik, visszatérési értéke `Query` típusú, és az SQL lekérdezés eredményét tartalmazza, mely az összes a rendszerben szereplő felhasználó felhasználóneve.

isAdmin

```
remote boolean isAdmin ( )
```

Output: enabled

Az `isAdmin` függvény segítségével ellenőrizhető, hogy az aktuális felhasználó rendelkezik-e adminisztrátori jogokkal. Argumentummal nem rendelkezik, mivel minden esetben az aktuális felhasználó adatait dolgozza fel. Visszatérési értéke `Boolean` típusú, „igaz” értékkel rendelkezik, amennyiben az aktuális felhasználó adminisztrátor.

saveUserData

```
remote void saveUserData ( required string oldName, required string userName, required string realName, required string password )
```

Output: suppressed
Parameters:
oldName: string, required, Aktuális felhasználónév
userName: string, required, Új felhasználónév
realName: string, required, Új valódi név
password: string, required, Új jelszó

A `saveUserData` függvény hivatott a felhasználó adatainak módosítása esetén szükséges táblamódosítások elvégzésére. Paraméterként a felhasználó aktuális és új felhasználónevét, új valódi nevét, illetve új jelszavát kapja meg. Visszatérési értékkel nem rendelkezik.

3.1.2.2 A „letters” komponens

A komponens feladata az üzenetek kezelése, az adatbázis „letters” és „switch” tábláinak kezelése (lekérdezések és módosítások). Ez a komponens is kizárólag függvényeket tartalmaz. A következőkben a „letters.cfc” komponens ezen alkotóelemeit mutatom be:

deleteLetter

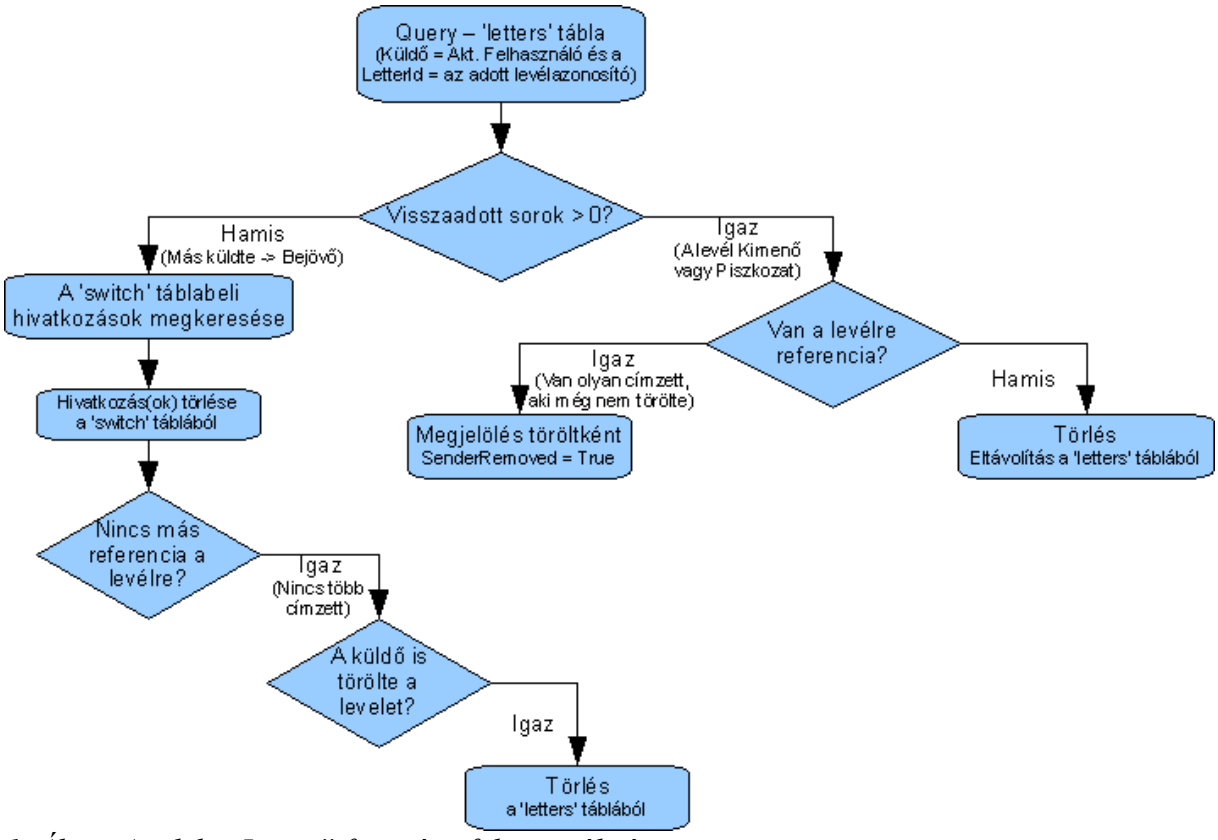
```
remote boolean deleteLetter ( required numeric letterID )
```

Output: enabled
Parameters:
letterID: numeric, required, Levélazonosító

A `deleteLetter` függvény segítségével törölhető egy adott levél a rendszerből. Paramétere az adott levélazonosító, visszatérési értéke `Boolean` típusú, igaz értékkel jelezve, hogy sikerült-e a törlési művelet. A komponensen belül ez a legösszetettebb függvény, mivel a levelek tárolásáért felelős két tábla felépítése komplikálttá teszi a törlési folyamatot. Ez a táblák tervezése folyamán kiküszöbölhető lett volna, de mivel ezen problémára csak a függvény írása során derült fény, a táblák módosítása rengeteg munkát vont volna maga után a többi függvényben szükségessé váló módosítások miatt. A függvény bonyolultságát fokozza, hogy több levéltípus (elküldött, beérkezett, piszkozat) törlését képes

elvégezni.

A függvény összetettsége jól látható a következő egyszerűsített folyamatábrán:



1. Ábra: A „deleteLetter" függvény folyamatábrája

getDraftLetters

```
remote query getDraftLetters ( required string
senderUserName )
```

Output: enabled
Parameters:
senderUserName: string, required, Küldő felhasználóneve

A `getDraftLetters` azon leveleket adja vissza, melyeket a felhasználó mentett piszkozatként. Paraméterként az adott felhasználó felhasználónevét kell megkapja, visszatérési értéke pedig `Query` típus lévén, egy összetett SQL lekérdezés eredményét adja vissza, melyben a levelek tulajdonságai is szerepelnek.

getIncomingLetters

```
remote query getIncomingLetters ( required string userName )
```

Output: enabled
Parameters:
userName: string, required, Felhasználónév

A `getIncomingLetters` a megadott felhasználónak címzett beérkező leveleket adja vissza. Paramétere ezen függvénynek is a felhasználónév, visszatérési értéke pedig szintén `Query` típus, s itt is az SQL lekérdezés eredményét tartalmazza.

getOutgoingLetters

```
remote query getOutgoingLetters ( required string
senderUserName )
```

Output: enabled
Parameters:
senderUserName: string, required, Küldő felhasználóneve

A `getOutGoingLetters` az előző függvény párjaként az adott felhasználóhoz tartozó kimenő levelek lekérdezésére szolgál. Paramétere ismét csak a felhasználónév, visszatérési értéke pedig ugyancsak `Query` típusként adja vissza a leveleket.

markAsRead

```
remote boolean markAsRead ( required numeric rowID )
```

Output: enabled
Parameters:
rowID: numeric, required, A "switch" táblabeli sorazonosító

A `markAsRead` függvény feladata az elolvasott levelek adatbázisban történő megjelölése a „switch” tábla megfelelő „read” mezőjének aktuális dátum-idő kombinációra történő állításával. Paramétere a megjelölendő levél „switch” táblabeli sorazonosítója. Visszatérési értéke `Boolean`, mely „igaz” értékkel jelzi a függvény sikeres végrehajtását.

sendLetter

```
remote numeric sendLetter ( required string userName, required
string letter, required array recipients, required string
subject )
```

Output: enabled
Parameters:
userName: string, required, Küldő felhasználóneve
letter: string, required, Az üzenet szövege
recipients: array, required, A címzettek valódi neve
subject: string, required, Az üzenet tárgya

A `sendLetter` függvény segítségével küldhető el egy levél. Paraméterei a küldő felhasználó, a levél szövege, a címzettek valódi neve egy tömb formájában, valamint a levél tárgya. Visszatérési értéke az elküldött levél levélazonosítója.

3.1.3 ColdSpring – AOP kiegészítés

Mint azt már korábban említettem, az alkalmazások esetében gyakran jelentkeznek átmetsző követelmények. Ezt jelen esetben az alkalmazásban a naplózási funkcióval próbálom demonstrálni. Ez tökéletes példa, mivel főleg a többfelhasználós üzleti alkalmazások esetében az egyik alapvető nemfunkcionális követelmény szokott lenni. Ennek megvalósítása viszonylag egyszerű lett volna a szerver-alkalmazás nyelvén, egy újabb komponens létrehozásával. Ezzel azonban át kellett volna hágni az objektum-orientált programozás szabályait, s megsértve az egységbezárás elvét, az új komponensből minden szükséges helyen egy új példányt létrehozni.

A funkció így aspektusorientált nyelven került megvalósításra, mely tekintve a szerver-alkalmazás nyelvét és fejlesztőeszközeit, a ColdSpring Framework segítségével készült el.

Amint azt már az aspektusorientált programozást tárgyaló résznél bemutattam, ilyen esetben az új funkcionalitás megvalósítása nem igényli a meglévő komponensek, kódok módosítását. A naplózáshoz egy teljesen új komponens került létrehozásra, melyből egyetlen példányt kell létrehozni a szerver-alkalmazás indítása során. A naplózás aspektusként történő megvalósítása mellett a ColdSpring Framework egy másik funkcióját is demonstrálandó, a naplózási aspektus „letters.cfc”, illetve „users.cfc” komponensekhez történő csatolása után egy ún. *RemoteProxy*-t is létrehoztam az így kiegészített komponensekből.

A RemoteProxy segítségével megadható, hogy az eredeti komponensek mely függvényeit szeretnénk elérhetővé tenni a távoli hívások számára. Ily módon az alkalmazás indításakor új komponensek jönnek létre, s a kliens-alkalmazás már ezeket fogja meghívni. Így nem szükséges az eredeti komponensek függvényeinek `remote` attribútummal történő ellátása sem, sőt azok úgy is elhelyezhetők a szerveren, hogy azokhoz a távoli hozzáférés ne is legyen lehetséges. Ezáltal a szerver-alkalmazás még biztonságosabbá tehető, mivel az újonnan keletkező komponensek nem tartalmazzák az eredetiek kódját, mindössze az azokra irányuló hívások találhatóak bennük. Ha tehát egy illetéktelen felhasználó hozzá is férne ezen komponensek kódjához, a függvények nevein, argumentumain, visszatérési értékein és az eredeti komponensek nevein kívül semmilyen információhoz nem jutna.

Mivel azonban a RemoteProxy új komponenseket hoz létre, a kliens-alkalmazásban meg kellett tenni némi változtatást: a távoli hívásokért felelős objektumok, a RemoteObject-ek tulajdonságai között a hívás céljaként megjelölt helyeket kellett megváltoztatni, hogy azok az új komponensekre mutassanak. Ez azért volt szükséges, mert az új komponensek a szerver fájlrendszerében máshol helyezkednek el. A másik lehetséges megoldás az eredeti komponensek áthelyezése, és az új komponenseknek a régiek helyére történő létrehozása lett volna.

Fontos azonban megjegyezni, hogy a RemoteProxy-k létrehozása nélkül tényleg nem lett volna szükség semmilyen változtatásra a kliens-alkalmazásban sem, s ily módon az újonnan hozzáadott funkció minden szempontból teljesen transzparens lenne, létezése egyik alkalmazás kódjából sem lenne észrevehető.

A naplózási funkció létrehozása viszonylag egyszerű feladat volt, mely a ColdSpring Framework egyszerűségének, valamint az aspektusok megírásához szükséges nyelvnek (CFML) köszönhető. Először is a ColdSpring Framework konfigurálását kell elvégezni. Ehhez egy XML fájl létrehozása szükséges, ahol babok definiálásával hozhatóak létre a kívánt funkciók. Itt kellett létrehozni a naplózást végző, valamint a belőlük RemoteProxy-kat létrehozó babokat. Az itt létrehozott XML fájl felépítése egyszerű, terjedelme mindössze néhány sor. Példaképp bemutatom a „users” komponens naplózással történő kiegészítésére definiált babokat, illetve az ebből RemoteProxy-t létrehozó babot.

```
<bean id="loggingAdvice"
```

```
class="Messenger.components.LoggingAdvice" />
```

A `loggingAdvice` bab mutat arra a ColdFusion komponensre, melynek feladata a naplózás elvégzése lesz. Jelen esetben ez a „LoggingAdvice.cfc” komponens.

```
<bean id="loggingAdvisor"
class="coldspring.aop.support.NamedMethodPointcutAdvisor">
     <property name="advice">
          <ref bean="loggingAdvice" />
     </property>
     <property name="mappedNames">
          <value>*</value>
     </property>
</bean>
```

A `loggingAdvisor` bab feladata, hogy a `mappedNames` tulajdonságnál megadott függvény(ek)hez hozzárendelje az `advice` tulajdonságnál megadott babo(ka)t. Mivel a függvények helyén „*” került megadásra, az összes függvény esetében meghívásra fog kerülni az `advice` tulajdonságnál beállított `loggingAdvice` bab.

```
<bean id="userService"
class="coldspring.aop.framework.ProxyFactoryBean">
     <property name="target">
          <bean class="Messenger.components.users" />
     </property>
     <property name="interceptorNames">
          <list>
               <value>loggingAdvisor</value>
          </list>
     </property>
</bean>
```

A `userService` bab `target` tulajdonságánál megtalálható komponenshez fognak hozzárendelődni az `interceptorNames` attribútumnál megadott babok. Így tehát a `users` komponenshez érkező hívás esetén, a `loggingAdvisor` beállításainak megfelelően, bármely függvény hívására a `loggingAdvice` bab kerül lefuttatásra.

```
<bean id="userServiceRemote"
class="coldspring.aop.framework.RemoteFactoryBean">
     <property name="target">
          <ref bean="userService" />
     </property>
     <property name="serviceName">
          <value>userServiceRemote</value>
     </property>
```

```
        <property name="beanFactoryName">
             <value>beanFactory</value>
        </property>
        <property name="relativePath">
             <value>/Messenger/components/remote/</value>
        </property>
        <property name="remoteMethodNames">
             <value>*</value>
        </property>
   </bean>
```

A `userServiceRemote` bab végzi el a korábban megvalósított naplózás RemoteProxy-ként történő exportálását. A `target` tulajdonságnál megadott érték jelzi, hogy mely babot kívánjuk exportálni. Ez jelen esetben a `userService` bab, mely a `users` komponenst már a naplózással kiegészítve tartalmazza. A `serviceName` jelzi a létrehozni kívánt proxy nevét, míg a `relativePath` annak fájlrendszerbéli helyét. Így tehát az új komponens a „/Messenger/components/remote/” helyen létrejövő „userServiceRemote.cfc” lesz. A `remoteMethodNames` értéke („*”) jelzi, hogy az összes függvényt exportálni kívánjuk.

A konfigurációs állomány létrehozását követően szükség van az abban található beállítások életbe léptetésére. Ehhez el kellett készítenem egy „Application.cfm” nevű speciális ColdFusion fájlt. Ez azért speciális, mert bármely CF komponens hívása esetén a CF szerver megkeresi az adott komponenshez a fájlrendszer hierarchiájában a gyökér irányában hozzá legközelebb eső `Application` komponenst vagy template-et („.cfc” vagy „.cfm” fájl), s az is lefuttatásra kerül.

Ily módon létrehozhatóak olyan függvények, melyek bármely komponens hívása esetén lefutnak, kezelhetővé válnak speciális, kifejezetten az alkalmazáshoz kötődő események (pl.: alkalmazás indítása, vége, általános kivételek, munkamenet-kezelés), alkalmazhatóak globális változók. Éppen ezért itt került létrehozásra a már korábban említett `Application.DSN` globális változó, illetve a munkamenet-kezelés is itt kapott helyet.

Habár a ColdSpring Framework inicializálásáért felelős utasítások elhelyezése az „Application.cfm” fájl `onApplicationStart` eseménykezelő függvényében lett volna evidens, az mégis egy különálló blokkba került. Ennek a fejlesztési folyamat szempontjából volt fontos szerepe, mivel az eseménykezelő függvényben történő elhelyezés esetén ez a

kódrészlet kizárólag az alkalmazás indításakor (vagyis a legelső olyan komponenshez történő híváskor, mely az adott „Application.cfm”-hez tartozik) került volna lefuttatásra. Ez esetben minden egyes módosítás után újra kellett volna indítani a ColdFusion szervert, ami nagyban növelte volna a fejlesztési időt.

Az inicializálást végző kódrészlet így a következő blokkon belül kapott helyet:

```
<cfif not structKeyExists(application,"beanFactory")
     or structKeyExists(url,"reloadApp")>
     <!--- Framework Inicializálás --->
</cfif>
```

Ezáltal a kódrészlet lefutása a következő két esetben következik be:

1. Az `Application.beanFactory` objektum nem létezik. Ez az objektum a ColdSpring Framework inicializálás során létrehozott, az alkalmazásban használt példánya.
2. A webszerverhez érkező GET parancs URL-je tartalmaz `reloadApp` értéket.

Ahhoz tehát, hogy a második opció nyújtotta lehetőséget használhassuk, szükség volt egy olyan weblap készítésére, melynek lekérésekor paraméterként megadható a `reloadApp`. Ez a weblap lett a már korábban említett „index.cfm”, mely ezen funkció mellett a keretrendszer működését is demonstrálja.

Az aspektusorientált rész másik fontos eleme a „LoggingAdvice.cfc” komponens, melyet már a ColdSpring konfigurálásához szükséges XML fájl bemutatásánál említettem, a naplózás elvégzéséhez szükséges kódot tartalmazza. Felépítését tekintve egyszerű, két függvényt tartalmaz:

1. `init` – Konstruktorként szolgál. A ColdSpring Framework használata miatt szükséges, meghívásakor a az adott komponenst példányosítja.
2. `InvokeMethod` – A ColdSpring Framework ezt a függvényt hívja meg minden egyes olyan esetben, ahol olyan függvény kerül lefuttatásra, melyhez hozzárendeltük azt a babot, ami ezt a komponenst tartalmazza tanácsként.

Mivel a komponens fő funkcióját az `invokeMethod` függvényben elhelyezkedő kód adja, ezt kívánom itt részletesen bemutatni.

Amennyiben egy olyan függvény kerül meghívásra, melyhez teendőként ez a komponens lett rendelve, a hívás nem közvetlenül a hívott függvényhez érkezik, hanem ezen komponens `invokeMethod` függvénye kerül meghívásra, s paraméterként kapja meg az eredetileg meghívott függvény nevét, s argumentumait. Ezáltal itt, a függvényen belül hozzáférhetővé válnak ezen információk, s lehetőség nyílik az „előtt”, „helyett” és „után” teendők megvalósítására. A függvényen belül elhelyezkedő kód feladata ugyanis, hogy elvégezze az eredetileg meghívni kívánt függvény lefuttatását. Így tehát amellett, hogy a hívás előtt vagy után helyezünk el egyéb teendőket, lehetőség nyílik arra is, hogy eldöntsük, végre kívánjuk-e hajtani a hívást. Amennyiben a hívást végrehajtjuk, annak visszatérési értékét is felhasználhatjuk a függvényen belül, de ilyen esetben visszatérési értékként itt is a végrehajtott hívás visszatérési értéket kell visszaadnunk.

A naplózás elkészítéséhez az „után” tanácsot megvalósító kódot hoztam létre, ily módon lehetővé téve, hogy a naplóbejegyzések során a meghívott függvények eredményét is felhasználhassam, mivel ez több esetben is fontos. Gondoljunk csak a „letters” komponens `sendLetter` függvényére, mely az elküldött levél azonosítójával tér vissza! Ezáltal nem csak az kerülhet naplózásra, hogy egy felhasználó elküldött egy levelet, de az is, hogy melyik levél lett elküldve, mely már ténylegesen hasznos információt hordoz.

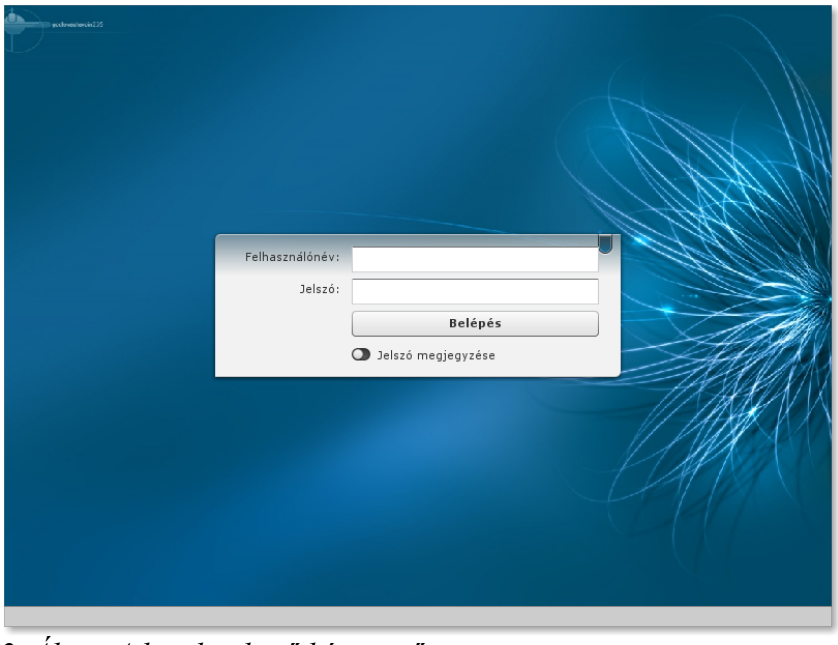
Mivel ez a komponens a „users” és „letters” komponensek mindegyik függvényéhez hozzá lett rendelve, fontos eldönteni, hogy mikor szükséges a naplózás. Ezt egy `switch` struktúrával oldottam meg, mely az eredetileg meghívott függvény nevét kiértékelve csak bizonyos függvények hívásához rendel (más és más) naplózási funkciót. Teljesen felesleges lenne – emellett pedig számottevő tárhelyet foglalna le az adatbázisban – ugyanis például a bejövő levelek lekérdezésének naplózása.

3.2 Kliens-oldali alkalmazás

A kliens-alkalmazás elkészítése Abobe Flex Builder 3 és az ehhez társítható Adobe AIR SDK segítségével készült, MXML és ActionScript 3 nyelveken.

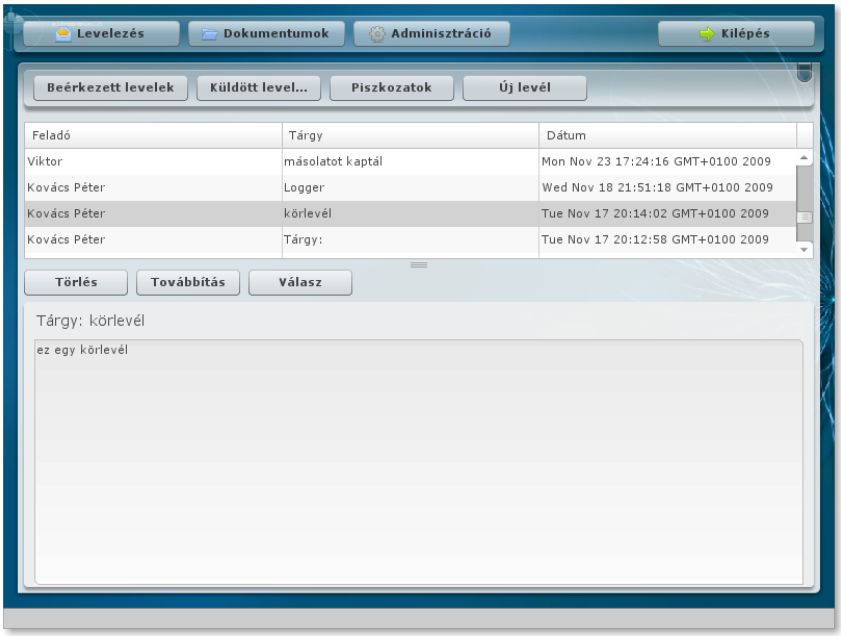
Az alkalmazás grafikus felhasználói felületének (angolul: GUI – Graphical User Interface) felépítéséhez az MXML-t alkalmaztam, mivel az jól strukturált, használata egyszerű, az így létrejövő kód áttekinthető. A felületen található objektumok elhelyezése így a

Flex Builder alkalmazásból, grafikus úton oldható meg. Ezáltal a fejlesztési folyamat ezen részére szánt idő nagy mértékben csökkenthető.



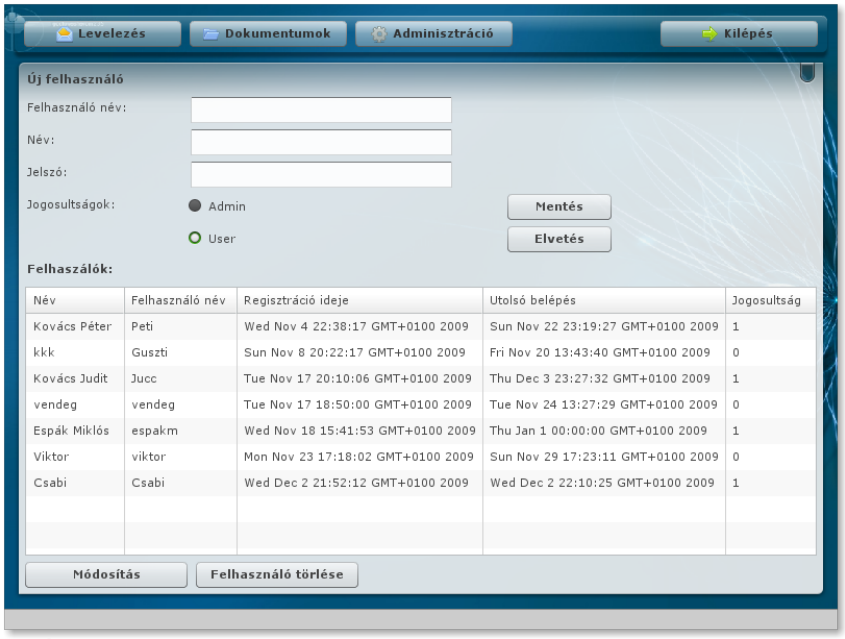
2. Ábra: A bejelentkező képernyő

A 2. ábrán látható a bejelentkező képernyő felépítése. Itt történik meg a felhasználók autentikációja.



3. Ábra: A beérkezett levelek

A 3. ábra mutatja a beérkezett levelek megtekintéséhez készített felületet, valamint a felső részen elhelyezkedő navigációs sávot, mellyel az alkalmazás többi funkcióját is elérhetjük.



4. Ábra: Az adminisztrációs felület

A 4. ábrán látható az adminisztrációs képernyő. Itt lehetséges a felhasználók adatainak megtekintése és módosítása, valamint a felhasználók regisztrálása és törlése. Az is látható, hogy mely felhasználó mikor jelentkezett be utoljára.

Az eseménykezelő és néhány minimális logikát megvalósító függvények kerültek ActionScript nyelven megvalósításra. Ez az Adobe fejlesztők között már régóta elterjedt, jól ismert, magas szintű objektum-orientált nyelv, mely leginkább a Java-hoz hasonlítható.

Az ActionScript használata segítségével tovább rövidíthető a fejlesztés folyamata. Feleslegessé válik ugyanis olyan folyamatok, függvények implementálása, melyek adott felhasználói felületen elhelyezkedő objektumok által megjelenített adatok frissítését vagy beállítását végzik. Amennyiben egy AS változót deklarálásakor ellátunk a `[Bindable]` attribútummal, lehetővé válik annak értékének hozzárendelése egyes GUI objektumokhoz. Ennek hatására ha egy ilyen változó értékét az AS kódon belül megváltoztatjuk, a változás azonnal megjelenik a felhasználói felületen is.

A Flex segítségével a távoli aszinkron hívások megvalósítása is egyszerű feladat. Mindössze a beépített `RemoteObject` (távoli objektum) objektumot kell példányosítanunk, annak attribútumait beállítanunk, s megadni a szerver-komponens elérni kívánt függvényeit. Példaképp bemutatom egy ilyen távoli hívás használatát:

```
<mx:RemoteObject
     id="roLetterService"
     destination="ColdFusion"
     source="Messenger.components.remote.letterServiceRemote"
     fault="faultHandler(event)" showBusyCursor="true"
     endpoint="http://petersmith.ddns.us/flex2gateway/">
     <mx:method
          name="getIncomingLetters"
          result="lettersResult(event)"
          fault="faultHandler(event)" />
     <mx:method
          name="getOutgoingLetters"
          result="lettersResult(event)"
          fault="faultHandler(event)" />
</mx:RemoteObject>
```

Az MXML kódrészlet felépítését tekintve egyszerű, jól átlátható. Egy új `RemoteObject` kerül létrehozásra, melynek `source` attribútuma jelzi, hogy a szerveren hol helyezkedik el a hívni kívánt komponens. Ez jelen esetben a már `RemoteProxy` által létrehozott, naplózással kiegészített `letterServiceRemote` komponens. Az `endpoint` definiálja a webszerver azon pontját, mely a hívások ColdFusion szerverhez történő átirányításáért felelős. Két meghívható függvényt jeleznek a `method` objektumok. Attribútumaik a `name`, mely a ColdFusion komponens egy függvényét jelöli, a `result` és a `fault`, melyek eseménykezelő függvényeket jelölnek. Előbbi a visszatérési érték megjelenésekor, utóbbi hiba keletkezése esetén kerül meghívásra.

Ezen kódrészlet megírását követően a távoli komponens beérkezett levelek lekérdezését végző függvénye a következőképp hívható meg az ActionScript kódból:

```
roLetterService.getIncomingLetters(MyGlobals.currentUser);
```

Az attribútum egy globális változó, mely az aktuális bejelentkezett felhasználó nevét tartalmazza.

A bejövő levelek megjelenítéséhez a visszatérési értéket kezelő függvényben mindössze annyi kódot kell elhelyezni, mely beállítja a korábban definiált – a leveleket tartalmazó – változó értékét:

```
[Bindable] private var letterList:ArrayCollection;
public function lettersResult(event:ResultEvent):void{
     letterList = event.result as ArrayCollection;
}
```

4. Az alkalmazás értékelése

Az alkalmazás elkészítése, és a teszt-periódus lezárása után elmondható, hogy egy olyan alkalmazást sikerült létrehoznom, mely jól mutatja a napjainkban egyre inkább elterjedőben lévő webalkalmazások felépítését, azok működését, s jól demonstrálja a diplomamunkám témájának ilyen területen történő alkalmazhatóságát.

4.1 A megvalósított funkciók tesztelése

A tesztelés folyamata három szakaszra bontható:

1. A programfejlesztés során a megvalósított funkciók folyamatos tesztelése zajlott.
2. Az alkalmazás elkészítését követően intenzív teszteket hajtottam végre.
3. A saját tesztelés elvégzését követően az élesben történő kipróbálás következett.

Természetesen, mint az minden szoftver fejlesztése során megszokott, a tesztelési folyamat már a kód írása során megkezdődött. Minden egyes függvény funkcionalitása ellenőrzésre került az elkészítését követően. Az alkalmazás elkészültekor saját magam végeztem intenzív tesztelést, mely során a nagyobb működésbeli hibákra fény derült, s természetesen ezek javításra is kerültek. Ezt követően néhány ismerősöm bevonásával végeztem egy utolsó tesztet. Ennek célja a valós működés szimulálása volt, mivel így több felhasználó használhatta a szoftvert – interneten keresztül –, ahogy az egy üzleti alkalmazás esetében is történne.

A végső teszt során már nem derült fény komoly, a funkcionalitást is érintő hibákra, mindössze néhány kényelmi funkció hiánya jelentkezett. Ezek implementálására már nem került sor, miután a diplomamunkám témájának bemutathatóságát ezen funkciók nem érintették.

4.2 Továbbfejlesztési lehetőségek

Az alkalmazás fejlesztése során törekedtem arra, hogy annak későbbi fejlesztése könnyen kivitelezhető legyen. Ezáltal egy olyan szoftvert hoztam létre, mely könnyen részét képezi egy komplexebb alkalmazásnak, vagy a funkciók bővítésével önmaga is azzá válhat.

Implementálásra került a felhasználókezelés, valamint a levelezési funkció. A

felhasználói felület létrehozásakor azonban már elhelyeztem a szükséges komponenseket egy dokumentumok megosztását lehetővé tévő funkcionalitás megvalósításához is. Ennek implementálása így már kizárólag a kliens-alkalmazás néhány újabb eseménykezelő függvényének elkészítését, valamint egy újabb szerver-oldali komponens létrehozását igényelné, s a meglévő kódban nem lenne szükség változtatásra.

5. Köszönetnyilvánítás

Köszönettel tartozom témavezetőmnek, Espák Miklósnak a szakmai irányításért, a rengeteg hasznos tanácsért, folyamatos rendelkezésre állásért, az alkalmazások tervezésének folyamatába való bevezetésért, valamint azért a szemlélet elsajátításáért, ami elengedhetetlenül fontos a szoftverfejlesztési munka során.

Szintén köszönet illeti Kovács Pétert, amiért rendelkezésemre bocsátotta szerverét, segített annak konfigurálásában, mely segítségével a demonstrációs alkalmazásom elkészítettem, s a tesztelést valós, életszerű körülmények mellett végezhettem.

Köszönet illeti egyetemi Tanárnőimet és Tanáraimat, amiért rengeteg hasznos tudást adtak át az évek során, s nagy mértékben hozzájárultak gondolkodásmódom pozitív formálásához.

6. Irodalomjegyzék

[1] Aspect-oriented software development Weboldal. http://aosd.net

[2] Dijkstra EW., *The structure of the "THE" multiprogramming system*, , 1968

[3] Parnas D., *A technique for software module specifications with examples*, , 1972

[4] Myers G., *Software Reliability: Principles and Practices*, Wiley, 1976

[5] Kiczales G, et al., *Aspect-Oriented Programming*, Xerox Research Center, 1997

[6] AspectJ. http://www.aspectj.org

[7] Hyper/J. http://www.research.ibm.com/hyperspace/HyperJ/HyperJ.htm

[8] Hyper/J Home Weboldal. http://www.alphaworks.ibm.com/tech/hyperj

[9] James Rumbaugh et al., *Object-Oriented Modeling and Design*, Prentice Hall, 1991

[10] Joseph D. Gradecki, Nicholas Lesiecki, *Mastering AspectJ - Aspect-Oriented Programming in Java*, Wiley, 2003

[11] Vladimir O. Safonov, *Using Aspect-Oriented Programming for Trustworthy Software Development*, Wiley, 2008

[12] AspectWerkz. http://aspectwerkz.codehaus.org

[13] ColdSpring Framework for ColdFusion Weboldal. http://www.coldspringframework.org/

[14] Adobe ColdFusion Weboldal. http://www.adobe.com/products/coldfusion/

[15] ColdSpring Framework Documentation. http://www.coldspringframework.org/index.cfm/go/documentation

[16] Tanenbaum, A.S., *Modern Operating Systems. 2nd ed.*, Prentice Hall, 2001

[17] Coffman, E.G., Elphick, M.J., Shoshani, A., *System Deadlocks, Computing Surveys, vol. 3*, , 1971

[18] MySQL. http://www.mysql.com/

[19] MySQL 5 Reference Manual. http://dev.mysql.com/doc/refman/5.0/en/index.html