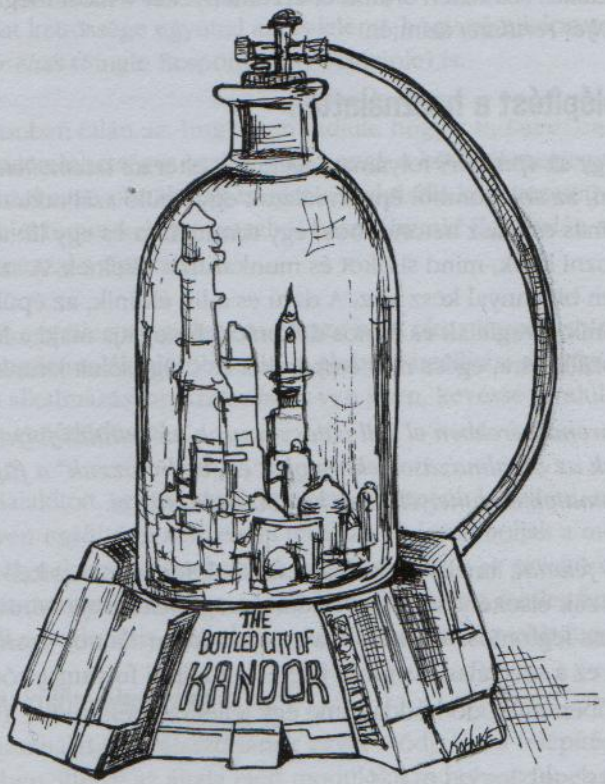
11

Rendszerek

Dr. Kevin Dean Wampler



„A bonyolultság lassú gyilkos. Kiszívja az életerőt a fejlesztőkből, és megnehezíti a termékek tervezését, felépítését és tesztelését."

– Ray Ozzie, műszaki igazgató, Microsoft Corporation

Hogyan építenénk fel egy várost?

Vajon boldogulnánk a részletekkel? Valószínűleg nem. Még egy kész város vezetése is meghaladja egyetlen ember képességeit. Persze a városok működnek valahogy (legalábbis az esetek többségében). Ez pedig azért lehetséges, mert szakemberek csoportjai dolgoznak a városirányítás egyes részfeladatain, így biztosítják a víz- és energiaellátást, a közlekedésszervezést, meghatározzák az építési szabályokat, és így tovább. Egyesek közülük az *összképért* felelnek, míg mások a részletekre összpontosítanak. A városok ugyanakkor kellően elvontak és modulárisak ahhoz, hogy az összetevők és az azokat irányító szakemberek anélkül is hatékonyan működjenek, hogy mindenkinek át kellene látnia a teljes képet.

A programfejlesztői csoportok gyakran ugyanígy szerveződnek, legfeljebb az elvonatkoztatási szintek eltérőek az általuk kezelt rendszerekben. A tiszta kóddal eddig az alacsonyabb elvonatkoztatási szinteken értünk el eredményeket – most megláthatjuk, miként boldoguljunk a *teljes rendszer* szintjén.

Válasszuk el a felépítést a használattól!

Vegyük észre, hogy az *építkezés* folyamata igencsak eltér az *üzemeltetéstől*. Miközben ezeket a sorokat írom, az ablakomból éppen rálátok egy épülő szállodára. Amit ma látok belőle, az egy hatalmas csupasz betondoboz, egy toronydaru és egy lift a munkásoknak. Akiket csak dolgozni látok, mind sisakot és munkaruhát viselnek. A szálloda azonban egy éven belül minden bizonnyal kész lesz. A daru és a lift eltűnik, az épület körüli összevisszaság megszűnik, üvegfalak és csinos dekoráció hívogatja majd a látogatókat. Ha ekkor nézünk rá a szállodára, egész más embereket és dolgozókat látunk majd.

> *A programrendszerekben el kell választanunk az indítási folyamatot (amelyben létrehozzuk az alkalmazásobjektumokat és „bedrótozzuk" a függőségeket) a futásidejű folyamatoktól, amelyek csak később indulnak el.*

Az indítás az első *feladat*, amellyel az alkalmazásnak foglalkoznia kell – ebben a fejezetben mi is ezt vesszük elsőként sorra. A feladatok vagy *felelősségi körök szétválasztása* az egyik legrégebbi és legfontosabb szabály a szakmánkban. Sajnos azonban az alkalmazások többségében ez a szétválasztás nincs jelen: az indítási folyamat kódja esetleges, és keveredik a futásidőben működő kóddal. Íme egy jellemző példa:

```
public Service getService() {
  if (service == null)
    service = new MyServiceImpl(...);   // Jó ez az alapértelmezés
                                        // a legtöbb esethez?
  return service;
}
```

Ez nem más, mint a *lusta előkészítés/kiértékelés* módszere, amely látszólag számos előnnyel rendelkezik – nem vesszük a vállunkra a felépítés költségét, hacsak nem használjuk valóban az objektumot, következésképpen a rendszerindítás összességében gyorsabb lesz. Mindemellett biztosítjuk, hogy soha ne adjunk vissza `null` értéket. Ugyanakkor a kódunk így függ a `MyServiceImpl` osztálytól és mindentől, amire ennek konstruktorában szükség van (amit most ügyesen kihagytunk). A függőségek kielégítése nélkül a programot nem tudjuk lefordítani – még akkor sem, ha futásidőben egyetlen ilyen típusú objektumot sem használunk.

A tesztelés is gondot okozhat. Ha a `MyServiceImpl` nehézsúlyú objektum, mindenképpen gondoskodnunk kell arról, hogy egy *tesztdublőrt*[1] (test double) vagy egy *álcaobjektumot* (mock object) a szolgáltatásmezőhöz rendeljünk, mielőtt a tagfüggvényt az egységteszt során meghívnánk. Mivel a felépítés kódja keveredik a futásidejű folyamatokkal, minden futási útvonalat meg kell vizsgálnunk (így például a `null` értéket vizsgáló kódblokkot). A feladat kettőssége egyúttal azt is jelenti, hogy némiképp megsértjük az *egyetlen felelősségi kör elvét* (Single Responsibility Principle) is.

A legrosszabb azonban talán az, hogy nem tudjuk, hogy a `MyServiceImpl` megfelel-e a céljainknak minden lehetséges esetben. Erre utalt a kódbeli megjegyzés is. Miért kell a tagfüggvényt tartalmazó osztálynak ismernie a globális környezetet? Vajon tudjuk-e majd valaha, melyik objektumot kellene igazából használnunk? Egyáltalán megoldható-e minden helyzet ugyanazzal a típussal?

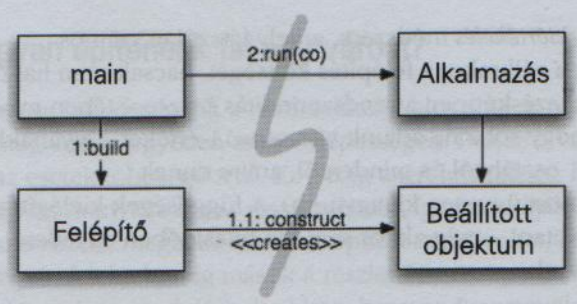
A *lusta előkészítés* egyetlen megjelenése persze nem okoz komolyabb problémát, de ahol egy van, ott jellemzően találunk többet is. A globális beállítási *stratégia* (ha van egyáltalán ilyen) így a teljes alkalmazásban *szétszórtan* van jelen, kevéssé moduláris formában, és gyakran rengeteg ismétlődéssel.

Ha *valóban* jól kialakított, szilárd rendszereket szeretnénk készíteni, nem szabad hagynunk, hogy az ilyen *csábítóan kényelmes* módszerek lerombolják a modularitást. Az indítási folyamat, amelynek során létrehozzuk és bedrótozzuk az objektumokat, nem kivétel ez alól. Ezt a folyamatot modulárisan el kell különítenünk a rendes futásidejű kódtól, és biztosítanunk kell egy globális, összehangolt módszert a nagyobb függőségek feloldására.

A main függvény különválasztása

A felépítés és a használat szétválasztásának egyik módja, ha a felépítés minden részletét a `main` függvényben, illetve az általa elért modulokban helyezzük el, a rendszer többi részét pedig azzal az előfeltételezéssel tervezzük meg, hogy a szükséges objektumokat már felépítettük és megfelelőképpen bedrótoztuk (lásd a 11.1. ábrát).

[1] [Mezzaros07]



11.1. ábra

A felépítés elhelyezése a main függvényben

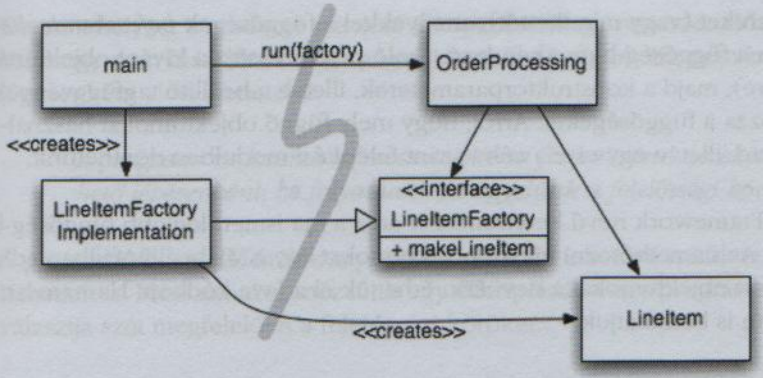
A vezérlés folyamatát itt egyszerű követni: a `main` függvény felépíti a rendszer működéséhez szükséges objektumokat, majd átadja azokat használatra az alkalmazásnak. Az objektumok tehát egy irányban haladnak – elfelé a `main` függvénytől. Figyeljük meg, hogy a függőség nyilai átmetszik a `main` függvény és az alkalmazás közötti határvonalat. Mindegyikük ugyanabba az irányba mutat: a `main` függvénytől elfelé. Ez azt jelenti, hogy az alkalmazásnak nem lehetnek ismeretei a `main` függvényről, illetve a felépítési folyamatról. Egyszerűen csak elvárja, hogy minden a legnagyobb rendben épüljön fel.

Gyárak

Előfordul, hogy egy alkalmazásnak abba is bele kell szólnia, hogy *mikor* hozunk létre egy adott objektumot. Egy rendeléseket feldolgozó programban például előbb létre kell hoznunk a megfelelő `LineItem` (tétel) példányokat, és csak ez után helyezhetjük el azokat egy `Order` (rendelés) objektumban. Ilyen esetekben használhatjuk az *elvont gyár*[2] mintát, ami lehetővé teszi az alkalmazásnak, hogy meghatározza a `LineItem` objektumok létrehozásának idejét, ugyanakkor elválasztja a felépítés részleteit az alkalmazáskódtól (lásd a 11.2. ábrát).

Ezúttal is megfigyelhetjük, hogy a függőségek iránya a `main` függvénytől az `OrderProcessing` alkalmazás felé mutat, vagyis az alkalmazást elválasztottuk a `LineItem` objektumok felépítésének részleteitől. Ezek a `LineItemFactoryImplementation` gyárban találhatók meg, tehát a `main` oldalán. Mindemellett a program pontosan szabályozhatja, hogy mikor szülessenek meg a `LineItem`-példányok, sőt még saját konstruktorparamétereket is megadhat.

[2] [GOF].



11.2. ábra

A felépítés leválasztása egy gyár segítségével

Függőség-befecskendezés

A felépítés és a használat elválasztásának hatékony módszere a *függőség-befecskendezés* (dependency injection, DI) is, ami voltaképpen a *vezérlés megfordításának* (inversion of control, IoC) alkalmazása a függőségek kezelésére.[3] Ezzel a módszerrel egy adott objektum másodlagos feladatait más, kifejezetten erre a célra szánt objektumra vagy objektumokra ruházzuk át, így megfelelhetünk az *egyetlen felelősségi kör* elvének. A függőségkezelés területén ez azt jelenti, hogy egy objektum nem vállalhatja magára a saját függőségei példányosításának feladatát. Ehelyett inkább átadja azt egy másik, felettes eljárásnak, megfordítva a vezérlés irányát. Mivel a beállítások elvégzése globális kérdés, ez a felettes eljárás többnyire a `main` függvényben vagy egy erre a célra létrehozott *tárolóban* kap helyet.

A JNDI-keresések a függőség-befecskendezés „részleges" megvalósításai, ahol egy objektum egy címtárkiszolgálótól kéri azt a „szolgáltatást", hogy keressen egyezést a megadott névvel:

```
MyService myService =
              (MyService)(jndiContext.lookup("NameOfMyService"));
```

A hívó objektumnak nincs befolyása arra, hogy milyen objektum érkezik majd válaszként (persze abban biztosak lehetünk, hogy megvalósítja a megfelelő felületet), de így is aktívan közrejátszik a függőség feloldásában.

A függőség-befecskendezés a tiszta alakjában ennél is tovább megy egy lépéssel. Ez esetben az osztály nem tesz közvetlen lépéseket a függőségei feloldására, hanem teljesen passzív: csupán elérhetővé teszi a megfelelő beállító tagfüggvényeket, illetve

[3] Lásd például: [Fowler].

konstruktorparamétereket (vagy mindkettőt), amelyekkel a függőségek *befecskendezhetők*. A felépítés során a függőség-befecskendező tároló példányosítja a kívánt objektumokat (többnyire kérésre), majd a konstruktorparaméterek, illetve a beállító tagfüggvények segítségével bedrótozza a függőségeket. Arról, hogy mely függő objektumokat használjuk, egy beállítófájlban, illetve egy erre a célra szánt felépítési modulban dönthetünk.

A Javához[4] a Spring Framework nevű keretrendszer adja a ma ismert legjobb függőség-befecskendező tárolót. Az összedrótozni kívánt objektumokat egy XML beállítófájlban adhatjuk meg, majd az adott objektumokat a nevükön érhetjük el a Java kódban. Hamarosan mindezt egy példában is bemutatjuk.

De mi a helyzet a *lusta előkészítés* előnyeivel? Nos, ez a módszer néha még a függőségek befecskendezésével együtt is hasznos. Először is, a legtöbb függőség-befecskendező tároló addig nem hozza létre az objektumokat, amíg szükség nincs rájuk. Másodszor, számos ilyen tároló lehetővé teszi gyárak használatát, illetve helyettesek létrehozását, amelyeknek a *lusta kiértékelés* és más hasonló optimalizálási lépések esetében vehetjük hasznát.[5]

A méretek növelése

A nagyvárosok kisvárosokból nőnek ki, a kisvárosok pedig falvakból. Az utak először szinte nem is léteznek, keskenyek, aztán burkolatot kapnak, és idővel kiszélesítik őket. A kisebb házak és az üres területek helyét nagyobb épületek foglalják el, ezek némelyike pedig átadja a helyét egy-egy felhőkarcolónak.

Kezdetben semmiféle szolgáltatás nem érhető el – villanyról, vízről, szennyvízelvezetésről vagy urambocsá' internetről ne is álmodjunk. Ahogy azonban a népesség és az épületek sűrűsége nő, szépen lassan hozzájutunk a szolgáltatások mindegyikéhez.

A növekedés azonban nem fájdalommentes. Gondoljunk csak bele, hányszor araszoltunk dugóban valamilyen útfelújítási projekt miatt, és morogtunk magunkban, hogy „Miért nem lehetett ezt eredetileg szélesebbre csinálni?"

Be kell azonban látnunk, hogy ez a dolgok rendes menete. Vajon ki adná az áldását egy hatsávos útra egy kisvároson keresztül, amely csak reménykedik a fejlődésben? Egyáltalán, ki *szeretne* ilyen utat a városán keresztül?

Hiú remény, hogy a rendszereket „elsőre tökéletesen" megtervezhetjük. Nem: ma csak a *jelen* elvárásainak felelhetünk meg, a jövő kérdéseit holnap kell kezelnünk a rendszer átalakításával és bővítésével. Lépésenként, folyamatosan növekedve haladjunk előre. Tesztvezérelt fejlesztés, átalakítás és tiszta kód – a kód szintjén ezek biztosítják a helyes irányt.

---

[4] Lásd: [Spring]. Létezik egy Spring.NET nevezetű környezet is.

[5] Ne feledjük, hogy a LUSTA KEZDETI ÉRTÉKADÁS/KIÉRTÉKELÉS mindössze egyszerű optimalizálás, ráadásul talán túl korai.

De mi a helyzet a rendszer szintjén? Elképzelhető egy rendszer előzetes tervezés nélkül? Valahogy nem hisszük el, hogy az egyszerűtől a bonyolultig nem kell mást tennünk, mint lépésenként előremenetelni.

> *A programrendszerek természete eltér a fizikai rendszerekétől. A felépítésük bővíthető lépésenként,* **ha** *folyamatosan ügyelünk a felelősségi körök szétválasztására.*

A programrendszerek múlékony természete teszi lehetővé mindezt, amint azt hamarosan láthatjuk is. Most azonban nézzünk egy ellenpéldát – egy programszerkezetet, amely nem választja szét megfelelően a felelősségi köröket.

Az eredeti EJB1 és EJB2 rendszerek nem választották szét megfelelően a feladatokat, így felesleges akadályokat állítottak a szerves növekedés elé. Figyeljük meg a `Bank` nevű *egyedbabszemet*. Az egyedbabszem (entity bean) nem más, mint relációs adatok memóriabeli alakja, vagyis egy tábla egy sora.

Először meg kell határoznunk egy helyi (folyamatbeli) vagy távoli (külön JVM alatt futó) felületet, amelyet az ügyfelek használhatnak. A 11.1. példában egy lehetséges helyi felületet mutatunk be:

11.1. példa

Helyi EJB2 felület a Bank EJB számára

```
package com.example.banking;
import java.util.Collections;
import javax.ejb.*;
public interface BankLocal extends java.ejb.EJBLocalObject {
  String getStreetAddr1() throws EJBException;
  String getStreetAddr2() throws EJBException;
  String getCity() throws EJBException;
  String getState() throws EJBException;
  String getZipCode() throws EJBException;
  void setStreetAddr1(String street1) throws EJBException;
  void setStreetAddr2(String street2) throws EJBException;
  void setCity(String city) throws EJBException;
  void setState(String state) throws EJBException;
  void setZipCode(String zip) throws EJBException;
  Collection getAccounts() throws EJBException;
  void setAccounts(Collection accounts) throws EJBException;
  void addAccount(AccountDTO accountDTO) throws EJBException;
}
```

Bemutattuk a `Bank` címének számos tulajdonságát, valamint a bank birtokában levő számlák gyűjteményét, amelyeknek az adatait külön `Account` EJB-k kezelik majd. A 11.2. példában a `Bank` babszem megfelelő megvalósítását mutatjuk be.

11.2. példa

A megfelelő EJB2 egyedbabszem-megvalósítás

```
package com.example.banking;
import java.util.Collections;
import javax.ejb.*;
public abstract class Bank implements javax.ejb.EntityBean {
  // Működési logika...
  public abstract String getStreetAddr1();
  public abstract String getStreetAddr2();
  public abstract String getCity();
  public abstract String getState();
  public abstract String getZipCode();
  public abstract void setStreetAddr1(String street1);
  public abstract void setStreetAddr2(String street2);
  public abstract void setCity(String city);
  public abstract void setState(String state);
  public abstract void setZipCode(String zip);
  public abstract Collection getAccounts();
  public abstract void setAccounts(Collection accounts);
  public void addAccount(AccountDTO accountDTO) {
    InitialContext context = new InitialContext();
    AccountHomeLocal accountHome =
                              context.lookup("AccountHomeLocal");
    AccountLocal account = accountHome.create(accountDTO);
    Collection accounts = getAccounts();
    accounts.add(account);
  }
  // Az EJB tároló kódja
  public abstract void setId(Integer id);
  public abstract Integer getId();
  public Integer ejbCreate(Integer id) { ... }
  public void ejbPostCreate(Integer id) { ... }
  // A többit meg kell valósítanunk, de többnyire üresek:
  public void setEntityContext(EntityContext ctx) {}
  public void unsetEntityContext() {}
  public void ejbActivate() {}
  public void ejbPassivate() {}
  public void ejbLoad() {}
  public void ejbStore() {}
  public void ejbRemove() {}
}
```

Az ide tartozó `LocalHome` felületet nem mutattuk be – ez lényegében egy objektumgyár. Nem mutattunk be egyet sem a `Bank` lehetséges kereső tagfüggvényei közül sem, amelyeket esetleg felhasználhatnánk.

Végezetül, meg kell írnunk néhány XML telepítésleírót, amelyek megadják az objektum–reláció leképezéseket, a kívánt tranzakciós viselkedést, a biztonsági korlátozásokat és más hasonlókat. A működési kód szorosan kapcsolódik az EJB2 „alkalmazástárolóhoz". A tárolótípusok alosztályait és a tároló működéséhez szükséges életcikluskezelő tagfüggvényeket magunknak kell elkészítenünk.

Mivel így egy nehézsúlyú tárolóhoz kötődünk, az elszigetelt egységtesztelés nehéz feladat. Ki kell váltanunk a tárolót, ami igen fáradságos, a másik lehetőség viszont az, hogy jelentős időt áldozunk arra, hogy az EJB2-t és a teszteket egy valódi kiszolgálóra telepítjük. Ez pedig valóban elpazarolt idő, hiszen a szoros csatolás miatt a kód használata az EJB2-n kívül gyakorlatilag lehetetlen.

Végezetül, ez a megoldás még magát az objektumközpontú programozást is aláássa. Egy babszem ugyanis nem örökölhet egy másiktól. Figyeljük csak meg az új számlák hozzáadásának kódját. Az EJB2-babszemek körében igen elterjedt módszer az „adatátviteli objektumok" (DTO-k) használata, amelyek lényegében „struktúrák", mindenféle viselkedés nélkül. Mindez jobbára feleslegesen ismétlődő adattípusok megjelenéséhez vezet, amelyek lényegében ugyanazokat az adatokat tartalmazzák, ráadásul ahhoz, hogy átmásoljuk őket egyik objektumból a másikba, még valamilyen segédkódra is szükség van.

Egymást átfedő felelősségi körök

Az EJB2 bizonyos területeken valóban képes szétválasztani a felelősségi köröket. Így például a kívánt tranzakciós, biztonsági, valamint egyes tároló műveleteket a telepítésleírókban, a forráskódtól függetlenül határozhatjuk meg. Vegyük észre, hogy az olyan *felelősségi körök*, mint a tárolás (maradandóság, persistance), jobbára áthágják az adott tartomány természetes objektumhatárait. Az objektumainkat azonos stratégiát követve szeretnénk tárolni, például meghatározott DBMS-t[6] használva egyszerű, „lapos" fájlok helyett, adott elnevezési szabályokat alkalmazva a táblákra és oszlopokra, egységes tranzakciós alakokat alkalmazva, és így tovább.

Lényegében modulárisan, az egységbe zárást szem előtt tartva tervezzük meg a tárolási stratégiát, a gyakorlatban mégis rengeteg objektumban kell gyakorlatilag ugyanazt a kódot felhasználnunk a maradandóság biztosítására. Az ilyen feladatokat nevezzük *egymást átfedő felelősségi köröknek*. Ismét leszögezzük, hogy maga a tárolási környezet és a tartománylogika külön-külön lehet tökéletesen moduláris – a gondot az okozza, hogy ezek az apró részletekben *átfedik* egymást.

Valójában az EJB olyan módon kezelte a tárolás, a biztonság és a tranzakció kérdéseit, hogy az már előrevetítette az *aspektusközpontú programozás* (aspect oriented programming, AOP)[7] módszerét, amely általános érvénnyel próbálja visszaállítani a modularitást az ilyen egymást átfedő felelősségi körök esetében.

[6] Database Management System, adatbázis-kezelő rendszer.

[7] Az aspektusokról általánosságban az [AOSD] ad felvilágosítást, az AspectJ jellemzőiről pedig az [AspectJ], illetve [Coyler] ír bővebben.

Az AOP-ben az *aspektusnak* (szempont) nevezett moduláris szerkezetek adják meg, hogy mely pontokon kell egységesen megváltoztatnunk a rendszer viselkedését ahhoz, hogy egy adott feladat kezelését támogassuk. Ezeket a beállításokat valamilyen tömör leíró vagy programozástechnikai módon rögzíthetjük.

A tárolásnál maradva, meghatározhatjuk, hogy mely objektumokat és jellemzőket (illetve ezek *mintáit*) kell megőriznünk, majd a tárolási feladatokat átadhatjuk a tárolási környezetnek („maradandósági keretrendszernek", persistence framework). A viselkedés módosítását *nem beavatkozó* módon[8] végezzük a célkódon az AOP környezet révén. Lássunk most három aspektust, illetve ahhoz hasonló megoldást a Javában.

Java helyettesek

A Java helyettesek (proxy) egyszerű helyzetek megoldására alkalmasak – például tagfüggvények hívásainak beburkolására egyes objektumokban vagy osztályokban. Mindazonáltal a JDK-val kapott dinamikus helyettesek kizárólag felületekkel működnek. Az osztályok helyettesítéséhez valamilyen bájtkódkezelő könyvtárat kell használnunk, mint a CGLIB, az ASM vagy a Javassist.[9]

A 11.3. példa egy JDK helyettes vázát mutatja, amely a maradandóság támogatását biztosítja a `Bank` alkalmazásunk számára – itt mindössze a számlák listájának kiolvasó és beállító tagfüggvényeit érintjük.

11.3. példa

JDK helyettes

```
// Bank.java (a csomagneveket elhagytuk)
import java.utils.*;
// Egy bank elvont ábrázolása
public interface Bank {
  Collection<Account> getAccounts();
  void setAccounts(Collection<Account> accounts);
}
// BankImpl.java
import java.utils.*;
// Az elvont ábrázolást megvalósító POJO (Plain Old Java Object)
public class BankImpl implements Bank {
  private List<Account> accounts;
  public Collection<Account> getAccounts() {
```

folytatódik

[8] Vagyis nincs szükség a célkód kézi szerkesztésére.

[9] Lásd [CGLIB], [ASM] és [Javassist].

11.3. példa

JDK helyettes – folytatás

```
    return accounts;
  }
  public void setAccounts(Collection<Account> accounts) {
    this.accounts = new ArrayList<Account>();
    for (Account account: accounts) {
      this.accounts.add(account);
    }
  }
}
// BankProxyHandler.java
import java.lang.reflect.*;
import java.util.*;
// Az "InvocationHandler", amelyre a helyettes API-nak szüksége van
public class BankProxyHandler implements InvocationHandler {
  private Bank bank;
  public BankHandler (Bank bank) {
    this.bank = bank;
  }
  // Az InvocationHandler-ben meghatározott tagfüggvény
  public Object invoke(Object proxy, Method method, Object[] args)
      throws Throwable {
    String methodName = method.getName();
    if (methodName.equals("getAccounts")) {
      bank.setAccounts(getAccountsFromDatabase());
      return bank.getAccounts();
    } else if (methodName.equals("setAccounts")) {
      bank.setAccounts((Collection<Account>) args[0]);
      setAccountsToDatabase(bank.getAccounts());
      return null;
    } else {
      ...
    }
  }
  // Egy csomó részlet:
  protected Collection<Account> getAccountsFromDatabase() { ... }
  protected void setAccountsToDatabase(Collection<Account>
accounts) { ... }
}
// Valahol másutt...
Bank bank = (Bank) Proxy.newProxyInstance(
  Bank.class.getClassLoader(),
  new Class[] { Bank.class },
  new BankProxyHandler(new BankImpl()));
```

Meghatároztuk a `Bank` nevű felületet, amelyet a helyettesbe *burkolunk*, továbbá a `BankImpl` nevű POJO-t, amely a működési logikát adja. (A POJO-król hamarosan még szót ejtünk.)

A Proxy API-nak szüksége van egy `InvocationHandler` objektumra – ezen keresztül valósítja meg a `Bank` helyetteshez intézett tagfüggvényhívásait. `BankProxyHandler` objektumunk a Java Reflection API használatával megfelelteti az általános tagfüggvényhívásokat a `BankImpl` megfelelő tagfüggvényeinek, és így tovább.

Rengeteg itt a kód, és ráadásul a szerkezete is összetett, pedig az eset egyszerű.[10] Ha valamelyik bájtkódkezelő könyvtárat vennénk használatba, azzal sem lenne egyszerűbb dolgunk. A kód mennyisége és bonyolultsága a helyettesek egyik hátulütője – megnehezítik a tiszta kód készítését. Ráadásul a helyettesekkel nem tudunk a teljes rendszer működését befolyásoló „sarokpontokat" találni, márpedig egy igazi AOP-re épülő megoldáshoz erre lenne szükség.[11]

Tiszta Java AOP környezetek

Szerencsére a helyettes nagyobb részben segédeszközök használatával automatikusan kezelhető. Helyetteseket számos Java-környezet – így a Spring AOP és a JBoss AOP – alkalmaz belsőleg, így az aspektusokat tiszta Java kóddal[12] valósíthatjuk meg. A Springben a működési logikát POJO-k (Plain Old Java Object) felhasználásával készíthetjük el, amelyek teljes egészében a saját tartományukban maradnak – nem függnek vállalati környezetektől vagy más tartományoktól. Következésképpen a felépítésük egyszerűbb, és a tesztelésük is könnyebb feladat. Ez a viszonylagos egyszerűség megkönnyíti, hogy programunkat a felhasználók jelen igényeinek megfelelően alakítsuk, ugyanakkor folyamatosan fejleszthessük a később megjelenő igényeknek megfelelően.

Beépítjük az alkalmazás működéséhez szükséges környezetet, köztük az olyan egymást átfedő felelősségi köröket, mint a tárolás, a tranzakciók, a biztonság, a gyorsítótárak, a hibatűrés és más egyebek – mindehhez leíró beállítófájlokat, illetve API-kat használunk. Számos esetben valójában Spring vagy JBoss könyvtári aspektusokat adunk meg, ahol a környezet intézi a háttérmunkát Java helyettesek, illetve bájtkódkezelő könyvtárak révén, a felhasználó számára láthatatlanul. Ezek a meghatározások szabályozzák a függőség-befecskendező tárolót, amely létrehozza a fontosabb objektumok példányait, és szükség szerint összedrótozza azokat.

---

[10] A Proxy API felépítéséről és használatáról jó példákkal szolgál [Goetz].

[11] Az AOP-t gyakran összetévesztik a megvalósítására szolgáló olyan módszerekkel, mint a tagfüggvény-elfogás vagy a helyettesekkel megvalósított „burkolás". Az AOP rendszerek legnagyobb erénye, hogy képesek rendszerszintű műveleteket tömör és moduláris formában elérhetővé tenni.

[12] Lásd [Spring] és [JBoss]. A „tiszta Java" kifejezés arra utal, hogy nem használjuk az AspectJ lehetőségeit.

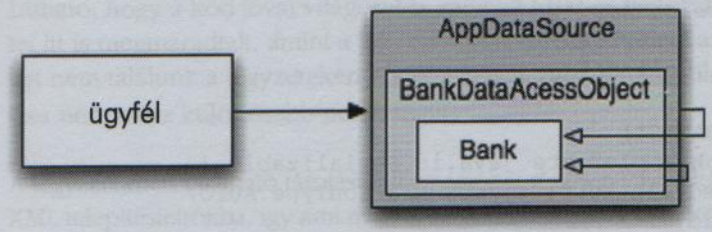
A 11.4. példában a Spring V2.5 `app.xml` beállítófájljának[13] egy jellemző részletét láthatjuk.

11.4. példa

Spring 2.X beállítófájl

```
<beans>
  ...
  <bean id="appDataSource"
  class="org.apache.commons.dbcp.BasicDataSource"
  destroy-method="close"
  p:driverClassName="com.mysql.jdbc.Driver"
  p:url="jdbc:mysql://localhost:3306/mydb"
  p:username="me"/>
  <bean id="bankDataAccessObject"
  class="com.example.banking.persistence.BankDataAccessObject"
  p:dataSource-ref="appDataSource"/>
  <bean id="bank"
  class="com.example.banking.model.Bank"
  p:dataAccessObject-ref="bankDataAccessObject"/>
  ...
</beans>
```

Az egyes babszemek egyfajta matrjoska-baba részei, amelyben a `Bank` tartományobjektumát egy adatelérési objektum (data access object, DAO) burkolja be (helyettesíti), amelyet ugyanakkor egy JDBC-meghajtó adatforrás helyettesít (lásd a 11.3. ábrát).



11.3. ábra

Matrjoskaként egymásba ágyazott díszítők

Az ügyfél azt hiszi, hogy a `getAccounts()` tagfüggvényt hívja meg egy `Bank` objektumra, pedig valójában azoknak az egymásba ágyazott *díszítő*[14] objektumoknak a legkülsőbbikét éri el, amelyek a `Bank` POJO alapviselkedését bővítik. Más díszítőket is üzembe állíthatunk a tranzakciók, a gyorsítótár és egyebek számára.

[13] A kód alapjául a `http://www.theserverside.com/tt/articles/article.tss?l=IntrotoSpring25` tartalma szolgált.

[14] [GOF].

Az alkalmazásban mindössze pár sorra van szükség ahhoz, hogy a függőség-befecskendező tárolótól elkérjük a rendszer legfelső szintű objektumait – ezt láthatjuk az XML fájlban:

```
XmlBeanFactory bf =
  new XmlBeanFactory(new ClassPathResource("app.xml", getClass()));
Bank bank = (Bank) bf.getBean("bank");
```

Mivel ilyen kevés, a Springhez köthető Java kódra van szükség, az alkalmazás csaknem teljesen független a Springtől, így nem jelentkeznek a szorosabb kapcsolatokat fenntartó rendszereknél (mint az EJB2) megszokott gondok.

Jóllehet az XML kód meglehetősen kiterjedtté és nehezen olvashatóvá is válhat[15], a beállítófájlokban így megadott „házirend" még mindig sokkal egyszerűbb, mint a háttérben automatikusan és láthatatlanul működő helyettesek és aspektusok rendszere. Ez a programszerkezet olyannyira sikeresnek bizonyult, hogy a Spring és hasonló környezetek hatására az EJB szabvány 3-as változatát teljesen átírták. Az EJB3 nagyjából a Springben megismert módon támogatja az egymást átfedő felelősségi körök beállítását XML fájlok, illetve a Java 5 jegyzetei segítségével.

A 11.5. példában a `Bank` objektum új változatát láthatjuk az EJB3-ban újraírva[16].

11.5. példa

A Bank osztály EJB3-as változata

```
package com.example.banking.model;
import javax.persistence.*;
import java.util.ArrayList;
import java.util.Collection;
@Entity
@Table(name = "BANKS")
public class Bank implements java.io.Serializable {
    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    private int id;
    @Embeddable // An object "inlined" in
    public class Address {
        protected String streetAddr1;
        protected String streetAddr2;
       protected String city;
        protected String state;
       protected String zipCode;
    }
```

folytatódik

[15] A példát leegyszerűsíthetjük olyan eljárások segítségével, amelyek a beállítások helyett a hagyományokra támaszkodnak, valamint Java 5 jegyzetek használatával, csökkentve a „bedrótozó" kód mennyiségét.

[16] A kód alapjául a `http://www.onjava.com/pub/a/onjava/2006/05/17/standardizing-with-ejb3-java-persistence-api.html` tartalma szolgált.

11.5. példa

A Bank osztály EJB3-as változata – folytatás

```
    @Embedded
    private Address address;
    @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER,
               mappedBy="bank")
    private Collection<Account> accounts = new
ArrayList<Account>();
    public int getId() {
       return id;
     }
     public void setId(int id) {
       this.id = id;
     }
    public void addAccount(Account account) {
       account.setBank(this);
       accounts.add(account);
     }
    public Collection<Account> getAccounts() {
       return accounts;
     }
    public void setAccounts(Collection<Account> accounts) {
       this.accounts = accounts;
    }
}
```

Látható, hogy a kód jóval világosabb, mint az EJB2-es változatban. Az egyed egyes részletei itt is megmaradtak, amint a jegyzetekben láthatjuk. Mivel azonban ilyen jellegű adatokat nem találunk a jegyzeteken kívül, a kód tisztának minősül, így a tesztelése és fenntartása nem okoz különösebb nehézséget.

A jegyzetekben található tárolási információkat részben vagy akár teljesen áthelyezhetjük XML telepítésleírókba, így ami marad, az valóban tiszta POJO kód. Ha a tárolás részletei nem változnak túl gyakran, sok fejlesztőcsoport megmarad a jegyzeteknél, de az EJB2 beavatkozó természetéhez képest még így is messze kevesebb hátulütővel kell szembesülnünk.

AspectJ-aspektusok

Végül ejtsünk szót az aspektusok kezelésének leghatékonyabb eszközéről, az AspectJ nyelvről[17] – ez a Java-bővítmény „első osztályú" támogatást ad a moduláris szerkezetek kiépítéséhez az aspektusok révén. A Spring AOP és a JBoss AOP tisztán Java megoldásai elegendőnek bizonyulnak az aspektusokat igénylő helyzetek 80–90 százalékában, de az

[17] Lásd [AspectJ] és [Colyer].

AspectJ használata így is megfontolandó, hiszen igen gazdag és hatékony eszköztárat biztosít a munkánkhoz. Hátránya, hogy számos új segédeszköz használatát el kell sajátítanunk, emellett új nyelvi szerkezetekkel is meg kell ismerkednünk.

Az átállási problémákat némiképp enyhíti az AspectJ nemrégiben bevezetett „jegyzetformátuma", amelyben Java 5 jegyzetek felhasználásával tiszta Java kódban határozhatjuk meg az aspektusokat. Emellett a Spring Framework számos olyan lehetőséggel rendelkezik, amelyek megkönnyítik a jegyzet alapú aspektusok befogadását az AspectJ használatában kevésbé járatos programozóknak.

Az AspectJ részletes tárgyalása meghaladná könyvünk kereteit. Erről a témakörről bővebben az [AspectJ], [Colyer] és [Spring] ad tájékoztatást.

Tesztvezérelt rendszerfelépítés

Nehéz eltúlozni a felelősségi körök aspektusokra – vagy hasonló szerkezetekre – épülő szétválasztásának jelentőségét. Egy rendszer *tesztvezérelt felépítése* csak akkor lehetséges, ha képesek vagyunk az alkalmazás tartománykódját POJO-kkal összeállítani, amelyek a kód szintjén nem kapcsolódnak a szerkezeti kérdésekhez. Így fejleszthetjük programunk szerkezetét az egyszerűtől az összetett felé, új eljárásokat beépítve, amennyiben erre szükség van. Nem feltétlenül kell egy „nagy előzetes tervet" (Big Design Up Front, BDUF)[18] készíteni. Sőt, az ilyen nagy tervek inkább hátráltatják a fejlődést, mivel meggátolják, hogy alkalmazkodjunk a változáshoz. Ennek leginkább az a pszichikai hatás az oka, amely szerint a változtatással feleslegessé válnának a korábbi erőfeszítéseink, illetve a szerkezeti döntések eleve meghatározzák, hogy milyen irányokba tapogatózhatunk.

Az építészek esetében a „nagy előzetes tervezés" az egyetlen járható út, mivel egy nagy fizikai szerkezet építése során már képtelenség gyökeres változtatásokat életbe léptetni.[19] A programoknak is van *fizikája*[20], de ha az alapfeladatokat megfelelően szétválasztjuk, *akkor* a gyökeres változtatások is gazdaságossá válhatnak.

Mindez azt jelenti, hogy egy szoftvert elkezdhetünk egy „naivan egyszerű", viszont szépen szétválasztott szerkezettel, amely megvalósítja a pillanatnyi felhasználói igényeket, és ezt bővíthetjük a későbbiekben. A világ legnagyobb webhelyei, amelyek igen magas rendelkezésre állást és teljesítményt nyújtanak kifinomult gyorsítótárakkal, virtualizációval, biztonsági és más hasonló lehetőségekkel, csak úgy juthattak el idáig, hogy a fejlesztőik a szerkezetben a lehető legkisebbre csökkentették az összetevők kapcsolatainak számát, továbbá minden elvonatkoztatási szinten és hatókörben kellően *egyszerű* felépítést alkalmaztak.

---

[18] Az előzetes tervezés egyébként jó módszer – most azonban arra a túlhajtott megoldásra utalunk, amikor mindent előre megtervezünk, mielőtt bármit is megvalósítanánk.

[19] Nem hallgathatjuk el, hogy a terv egészében és részleteiben ez esetben is változhat lépésről lépésre, még az építkezés megkezdése után is.

[20] A *szoftverfizika* kifejezést elsőként [Kolence] említi.

Mindez persze nem jelenti azt, hogy célok és irányok nélkül indulunk neki egy projektnek. Vannak elvárásaink a fejlesztés általános hatókörével, céljaival és ütemezésével kapcsolatban, és van képünk az eredményként kapott rendszerről is. Fontos azonban, hogy mindeközben megőrizzük a képességünket arra, hogy alkalmazkodjunk a körülmények változásaihoz.

Az EJB csak egy a számos API közül, amelyek túltervezettek, és ez igencsak aláássa a felelősségi körök szétválasztásának ügyét. Még a jól megtervezett API-k is túlzásnak bizonyulnak, ha nincs szükség a lehetőségeik többségére. A jó API nagyjában-egészében *láthatatlanná válik*, így a csapat a kreatív energiáit inkább a felhasználói igények kielégítésére fordíthatja. Ha nem így járnak el, a szerkezeti korlátok a felhasználó kiszolgálása ellen hatnak.

Mindezt röviden így foglalhatjuk össze:

> *Az optimális rendszerszerkezet moduláris feladattartományokból épül fel, amelyeket POJO-kkal vagy más objektumokkal valósítunk meg. A különböző tartományok közötti kölcsönhatásokat a kódba a lehető legkisebb mértékben beavatkozó aspektusok, illetve hozzájuk hasonló eszközök biztosítják. Ezt a rendszert már hatékonyan tesztelhetjük, hasonlóan a kódhoz.*

A döntéshozatal optimalizálása

A modularitás és a felelősségi körök szétválasztása lehetővé teszi, hogy eltávolodjunk a központosított döntéshozataltól. Egy elegendően nagy rendszerben – legyen szó nagyvárosról vagy egy kiterjedt programrendszerről – egyetlen személy nem képes érdemi döntéseket hozni.

Tudjuk, hogy a felelősségeket az adott területen legalkalmasabb emberek kezébe érdemes adnunk. Azt viszont gyakorta elfelejtjük, hogy *jobb, ha a döntéseket az utolsó lehetséges pillanatig halasztjuk el.* Ez nem lustaság vagy felelőtlenség – így tudjuk a döntésünket a lehető legteljesebb tudás birtokában meghozni. A korai döntés egyet jelent az elégtelen tudás alapján meghozott döntéssel. Így kevesebb visszajelzést kapunk az ügyfeleinktől, kevésbé látjuk át a projektet, és kevesebb tapasztalat birtokában döntünk.

> *A moduláris feladatokra bontott POJO-k rendszere lehetővé teszi, hogy a döntéseinket éppen a megfelelő időben hozzuk meg, a legfrissebb tudás birtokában. Így a döntéseink bonyolultsága is csökken.*

Csak akkor használjunk szabványokat, ha ez látható előnyökkel jár!

Az építkezéseken valóságos csodák történnek: csak lessük, milyen ütemben növekednek a háztömbök (még a tél közepén is), és milyen döbbenetes technikai megoldások lehetségesek a mai eszközeinkkel. Az építőipar kiforrott szakterület, a századok során egyre tökéletesített összetevőkkel, módszerekkel és szabványokkal.

Számos fejlesztőcsoport azért döntött az EJB2 mellett, mert az volt a szabvány – még akkor is, ha egyébként sokkal könnyebb súlyú és egyszerűbb megoldások is rendelkezésre álltak volna. Nem egy csapatot láttam, akik annyira belebolondultak a *felkapott* szabványokba, hogy eközben megfeledkeztek az ügyfeleik valódi igényeiről.

> *A szabványok megkönnyítik az ötletek és összetevők újrahasznosítását, a megfelelő tapasztalattal rendelkező szakemberek kiválasztását, a jó ötletek megvalósítását és a komponensek összedrótozását. Mindazonáltal a szabványok kidolgozása gyakran túl hosszú ideig tart ahhoz, hogy ezt a fejlesztők kivárják, és bizonyos szabványok elvesztik a kapcsolatot azokkal, akik valójában felhasználnák őket.*

A rendszereknek tartományfüggő nyelvekre van szükségük

Az építőipar, más tartományokhoz, szakterületekhez hasonlóan gazdag nyelvezetet alakított ki, saját szókinccsel, kifejezésekkel és mintákkal[21], amelyek a szükséges információkat tisztán és tömören képesek átadni. A programfejlesztés világában újabban ismét az érdeklődés középpontjába kerültek a *tartományfüggő nyelvek* (Domain Specific Language, DSL)[22], vagyis azok az önálló, apró parancsnyelvek, illetve API-k a szabványos programozási nyelvekben, amelyek lehetővé teszik, hogy olyan formában készítsük el a kódot, ami a tartomány szakemberei számára gyakorlatilag hétköznapi szövegként olvasható.

A jól megalkotott szaknyelv csökkenti a szakadékot a tartomány elve és az azt megvalósító kód között éppúgy, mint ahogy ügyes módszerekkel javíthatjuk a fejlesztőcsapat és a projekt gazdái közötti érintkezést. Ha a tartománylogikát ugyanazon a nyelven készítjük el, mint amit a tartomány szakemberei is használnak, kisebb a kockázata annak, hogy az elveket hibásan öntjük kódba. Ha megfelelőképpen használjuk a tartományfüggő nyelveket, az elvonatkoztatási szintet a kódszerkezetek és a tervezési minták fölé emelhetjük. Ezek teszik lehetővé a fejlesztőknek, hogy a kód rendeltetését a megfelelő elvonatkoztatási szinten fedjék fel.

> *A tartományfüggő nyelvek használatával az összes elvonatkoztatási szintet és tartományt POJO-kkal fejezhetjük ki, a legfelső szintű eljárásoktól az alacsony szintű részletekig.*

---

[21] [Alexander] munkája különösen nagy hatással volt a programozói társadalomra.

[22] Lásd például [DSL]. A [JMock] egy Java API-t mutat be, amely jól példázza, hogy miként hozhatunk létre tartományfüggő nyelvet ebben a formában.

Összefoglalás

Ügyelnünk kell a rendszerek tisztaságára is. Egy beavatkozó szerkezet eluralja a tartománylogikát, és hatással van a rendszer rugalmasságára. Ha a tartománykód homályos, azt a minőség sínyli meg, mivel a hibák könnyebben elrejtőzhetnek, és a felhasználói igényeket nehezebb megvalósítani. Ha a rugalmasság sérül, a hatékonyság is csökken, és a tesztvezérelt fejlesztés előnyei semmibe vesznek.

A szándékainkat tisztán ki kell fejeznünk minden elvonatkoztatási szinten. Ez csak úgy érhető el, ha POJO-kat készítünk, és az aspektusokhoz hasonló megoldásokkal nem beavatkozó módon építjük be az egyéb megvalósítási kérdéseket.

Akár rendszereket, akár modulokat tervezünk, soha ne feledjük, hogy a *legegyszerűbb, még működő módszert kell használnunk.*

Irodalomjegyzék

[Alexander]: Christopher Alexander: A *Timeless Way of Building*, Oxford University Press, New York, 1979.

[AOSD]: Aspect-Oriented Software Development port, `http://aosd.net`

[ASM]: Az ASM honlapja, `http://asm.objectweb.org/`

[AspectJ]: `http://eclipse.org/aspectj`

[CGLIB]: Code Generation Library, `http://cglib.sourceforge.net/`

[Colyer]: Adrian Colyer, Andy Clement, George Hurley, Mathew Webster: *Eclipse AspectJ*, Person Education, Inc., Upper Saddle River, NJ, 2005.

[DSL]: Domain-specific programming language `http://en.wikipedia.org/wiki/Domain_specific_programming_language`

[Fowler]: *Inversion of Control Containers and the Dependency Injection pattern*, `http://martinfowler.com/articles/injection.html`

[Goetz]: Brian Goetz: *Java Theory and Practice: Decorating with Dynamic Proxies*, `http://www.ibm.com/developerworks/java/library/j-jtp08305.html`

[Javassist]: A Javassist honlapja, `http://www.csg.is.titech.ac.jp/~chiba/javassist/`

[JBoss]: A JBoss honlapja, `http://jboss.org`

[JMock]: JMock – pehelysúlyú álcaobjektum-könyvtár a Javához, `http://jmock.org`

[Kolence]: Kenneth W. Kolence: *Software physics and computer performance measurements*, in: *Proceedings of the ACM annual conference – Volume 2*, Boston, Massachusetts, 1024–1040. oldal, 1972.

[Spring]: A Spring keretrendszer honlapja, `http://www.springframework.org`

[Mezzaros07]: Gerard Mezzaros: *XUnit Patterns*, Addison-Wesley, 2007.

[GOF]: Gamma és mások: *Design Patterns: Elements of Reusable Object Oriented Software*, Addison-Wesley, 1996 (magyarul: *Programtervezési minták – Újrahasznosítható elemek objektumközpontú programokhoz*, Kiskapu, 2004).