11 ARCHITEKTURÁLIS TERVEZÉS

TÉMAKÖRÖK

A fejezet célja megismertetni a szoftver architektúrájának és architekturális tervezésének fogalmait. Mire a fejezet végére érünk:

- meg fogjuk érteni, hogy miért fontos a szoftver architekturális tervezése;
- meg fogjuk érteni az architekturális tervezés folyamata során a rendszer architektúrájával kapcsolatban meghozandó döntéseket;
- megismerünk három egymást kiegészítő, a rendszer szerkezetét, vezérlését és moduláris felbontását lefedő rendszer-architektúra stílust;
- meg fogjuk érteni, hogy hogyan lehet referenciaarchitektúrák segítségével az architekturális fogalmakat kommunikálni és a rendszerarchitektúrákat értékelni.

TARTALOM

A nagy rendszereket olyan alrendszerekre bontják, amelyek biztosítják az egymással kapcsolatban lévő szolgáltatásokat. Ezen alrendszerek azonosításának és az alrendszer vezérlésére és kommunikációjára szolgáló keretrendszer létrehozásának kezdeti tervezési folyamatát architekturális tervezésnek, míg a tervezési folyamat kimenetét a szoftverarchitektúra leírásának nevezzük.

A 4. fejezetben bemutatott modellben a tervezési folyamat első lépése az architekturális tervezés, amely a tervezés és a követelménytervezés folyamatai közötti kritikus kapcsolatot reprezentálja. Az architekturális tervezés folyamata egy rendszer alapvető strukturális keretrendszerének kialakításával foglalkozik, amely azonosítja a rendszer fő komponenseit és az ezek közötti kommunikációt.

Bass és mások (Bass *és mások,* 2003) a szoftverarchitektúra explicit tervezésének és dokumentálásának három előnyét említi meg:

1. *Kulcsfigurák kommunikációja.* Az architektúra a rendszer olyan magas szintű bemutatása, amely alapjául szolgálhat a különböző kulcsfigurák rendszerről folytatott megbeszéléseinek.
2. *Rendszerelemzés.* A rendszer-architektúrának a rendszerfejlesztés korai szakaszában történő részletezése lehetőséget biztosít valamennyi elemzésre. Az architekturális tervezés döntései óriási hatással vannak arra, hogy a rendszer képes-e megfelelni a kritikus követelményeknek – mint amilyen például a teljesítmény, a megbízhatóság és a karbantarthatóság – vagy sem.
3. *Széles körű újrafelhasználhatóság.* A rendszer-architektúra modellje kompakt, kezelhető leírása annak, hogyan szerveződik a rendszer és hogy a komponensek hogyan működnek együtt. A rendszer-architektúra a hasonló követelményekkel rendelkező rendszerek esetén gyakran megegyezik, és így segíti a széles körű szoftver-újrafelhasználhatóságot. Ahogyan azt a 18. fejezetben tárgyalom, lehetőség van olyan termékvonal-architektúrák kifejlesztésére, ahol az egymással kapcsolatban álló rendszerek ugyanazt az architektúrát használják.

Hofmeister és mások (Hofmeister *és mások,* 2000) arról írtak, hogy az architekturális tervezési lépés hogyan kényszeríti rá a szoftvertervezőket, hogy a tervezés kulcsfontosságú aspektusaival már a folyamat korai szakaszában foglalkozzanak. Azt javasolják, hogy a szoftver architektúrája szolgáljon a tervezés terveként, amit a rendszerkövetelmények egyeztetésére és az ügyfelekkel, fejlesztőkkel és menedzserekkel történő megbeszélésekre egyaránt használhatunk. Ez a bonyolultság kezelésének fontos eszköze, mert elrejti a részleteket, és lehetővé teszi, hogy a tervezők a kulcsfontosságú rendszerabsztrakciókra koncentráljanak.

A rendszer architektúrája befolyásolja a rendszer teljesítményét, robusztusságát, eloszthatóságát és karbantarthatóságát (Bosch, 2000). Az alkalmazás számára választott szerkezet ezért nemfunkcionális rendszerkövetelményektől is függhet:

1. *Teljesítmény.* Ha a teljesítmény kritikus követelmény, az architektúrát úgy kell megtervezni, hogy a kritikus műveleteket kisszámú alrendszerbe helyezzük el, amelyek között a lehető legkevesebb kommunikáció zajlik. Ez viszonylag durva szemcsézettségű, nagyméretű komponensek kialakítását jelentheti.
2. *Védettség.* Ha a védettség kritikus követelmény, rétegzett architektúrára van szükség, ahol a leginkább kritikus eszközöket a legbelső rétegek védik, amelyekre magas szintű védettség-ellenőrzést kell végezni.
3. *Biztonságosság.* Ha a biztonságosság kritikus követelmény, az architektúrát úgy kell kialakítani, hogy a biztonságossághoz kapcsolódó műveletek egyetlen vagy kisszámú alrendszerben kapjanak helyet. Ez csökkenti a biztonságosság ellenőrzésének költségeit és gondjait, és lehetővé teszi, hogy kapcsolódó védelmi rendszereket biztosítsunk.
4. *Rendelkezésre állás.* Ha a rendelkezésre állás kritikus követelmény, az architektúra tervezése során redundáns komponenseket kell felvennünk, hogy lehetővé váljon a komponenseknek a rendszer leállítása nélkül történő cseréje,

illetve módosítása. A nagy rendelkezésre állással bíró rendszerek hibatűrő rendszer-architektúráit a 20. fejezet tárgyalja.

5. *Karbantarthatóság.* Ha a karbantarthatóság kritikus követelmény, akkor az architektúra tervezésekor finom szemcsézettségű, önálló komponenseket kell használnunk, amelyek könnyen változtathatók. Az adatokat előállító és azokat feldolgozó komponenseket külön kell választani, és el kell kerülni az osztott adatszerkezeteket.

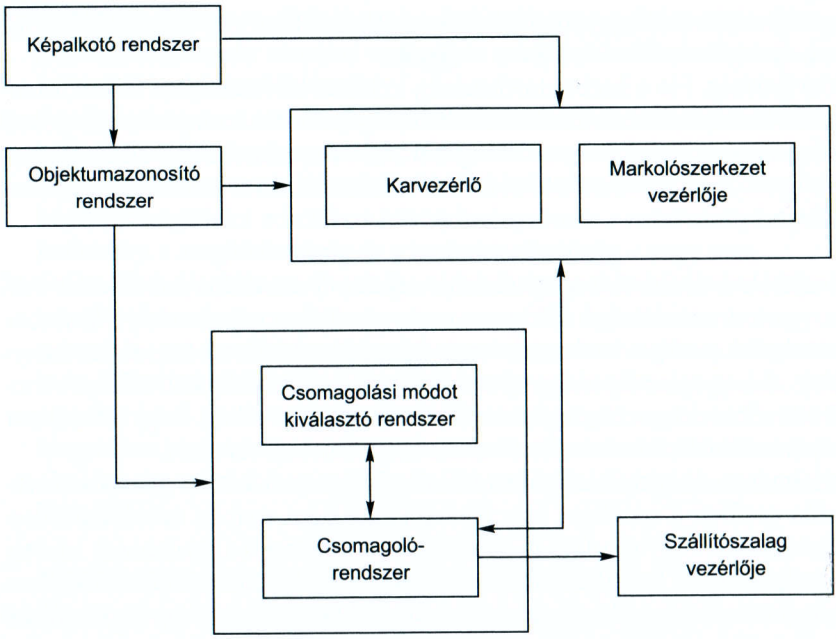
Ezen architektúrák között meghúzódik néhány potenciális konfliktus. Például a durva szemcsézettségű komponensek növelik a teljesítményt, a finom szemcsézettségűek pedig a karbantarthatóságot. Ha mindkettő fontos rendszerkövetelmény, akkor valamilyen kompromisszumos megoldást kell találni. Ahogyan arról a későbbiekben még írok, ezt például úgy lehet elérni, hogy a rendszer különböző részeit különböző architekturális stílusban kell tervezni.

A követelmény- és architekturális tervezés folyamatai között jelentős az átfedés. Ideális esetben a rendszer specifikációjának nem szabad tervezési információt tartalmaznia. A gyakorlatban azonban ez a nagyon kis rendszerek kivételével nem valósághű. A specifikáció strukturálásához architekturális felbontásra van szükség. Erre a 2. fejezetben hoztam példát, ahol a 2.8. ábrán egy légi irányító rendszer architektúrája látható. Egy ilyen architekturális modellt az alrendszer-specifikáció kiindulópontjaként használhatunk.

Az alrendszerek tervezése tulajdonképpen a rendszer durva szemcsézettségű komponensekre történő absztrakt felbontása, amely komponensek lehetnek önálló rendszerek. Az alrendszerek terveit általában blokkdiagram segítségével írjuk le, ahol a diagram dobozai az egyes alrendszereket reprezentálják. A dobozokon belüli dobozok azt jelzik, hogy az alrendszer újabb alrendszerekre bontható. A nyilak jelzik az adatok és/vagy a vezérlés irányát az egyes alrendszerek között. A blokkdiagramok a rendszer szerkezetének áttekintő képét adják, amely a rendszerfejlesztésben részt vevő különféle területekről érkezett emberek számára könnyen érthető.

A 11.1. ábrán egy automatizált csomagolórendszer architektúrájának absztrakt modellje van, és látható, hogy mely alrendszereket kell kifejleszteni. Ez a csomagolórendszer különféle objektumok csomagolására képes. Használ egy felismerő alrendszert, amely a futószalagról kiválasztja a csomagolandó tárgyakat, meghatározza azok típusát, majd eldönti, hogy melyik a megfelelő csomagolási módszer. Ezt követően leemeli az objektumot a futószalagról, elvégzi a csomagolást, majd a becsomagolt tárgyat egy másik futószalagra helyezi. Az ugyanezen szinten végzett architekturális tervezésre egyéb példák a 2. fejezetben (2.6. és 2.8. ábrák) láthatók.

Bass és mások (Bass *és mások*, 2003) azt állítják, hogy a dobozokból és a vonalakból álló egyszerű diagramok nem használhatók jól az architekturális reprezentációkra, mert nem mutatják meg sem a rendszerkomponensek közötti kapcsolatok természetét, sem pedig azok kívülről látható tulajdonságait. A rendszertervezők szempontjából ez teljesen igaz is, azonban az ilyenfajta modell a rendszer kulcsfiguráival történő kommunikáció, valamint projekttervezés esetén hatékony. Mi-



11.1. ábra. Egy csomagoló robot vezérlőrendszerének blokkdiagramja

vel nincs túlzsúfolva részletekkel, a kulcsfigurák kapcsolatba kerülhetnek vele és megérthetik a rendszer absztrakt nézetét. Azonosítja az egymástól függetlenül fejleszthető kulcsfontosságú alrendszereket, így a menedzserek elkezdhetik kijelölni azokat az embereket, akik ezeknek a rendszereknek a fejlesztését tervezik. A doboz- és vonaldiagramok természetesen nem az egyetlen alkalmazott architekturális reprezentációk, azonban úgy gondolom, hogy ez a diagramfajta egyike a használható architekturális modelleknek.

Annak eldöntése, hogy a rendszert hogyan kell alrendszerekre bontani, igen nehéz probléma. Természetesen a fő szempontot a rendszerkövetelmények jelentik, és olyan tervet kell készítenünk, ahol a követelmények és az alrendszerek igen közel állnak egymáshoz. Ekkor ugyanis a követelmény megváltozása esetén a változás lokalizálva van, nem pedig több alrendszer között oszlik meg. A 13. fejezetben bemutatok számos olyan alkalmazásarchitektúrát, amelyek kiindulópontként szolgálhatnak az alrendszerek azonosításában.

11.1. ARCHITEKTURÁLIS TERVEZÉSI DÖNTÉSEK

Az architekturális tervezés kreatív folyamat, ahol megpróbálunk előállítani egy rendszerszerkezetet, amely megfelel a funkcionális és nemfunkcionális rendszerkövetelményeknek. Mivel ez kreatív folyamat, az elvégzendő tevékenységek nagyban függnek a kifejlesztés alatt álló rendszer típusától, a rendszert építő

személy hátterétől és tapasztalatától, valamint a rendszer követelményeitől. Éppen ezért, az architekturális tervezési folyamatra hasznosabb döntési, mint tevékenységi szempontból tekintenünk. Az architekturális tervezési folyamat során a rendszer tervezőinek számos olyan döntést kell meghozniuk, amelyek alapvetően kihatnak a rendszerre és a fejlesztési folyamatra. Tudásuk és tapasztalataik alapján a következő alapvető kérdésekre kell válaszolniuk:

1. Létezik-e olyan általános alkalmazásarchitektúra, amely a tervezendő rendszer számára mintául szolgálhat?
2. Hogyan osztjuk szét a rendszert processzorokra?
3. Milyen architekturális stílus lenne a rendszer számára megfelelő?
4. Milyen alapvető megoldásokat alkalmazunk a rendszer strukturálására?
5. Hogyan lehet a rendszer szerkezeti egységeit modulokra felbontani?
6. Milyen stratégiát kell alkalmazni az egységek működésének vezérlésével kapcsolatban?
7. Hogyan értékelik majd ki az architekturális tervet?
8. Hogyan kell a rendszer-architektúrát dokumentálni?

Habár minden szoftver egyedi, ugyanazon szakterület alkalmazásai gyakran hasonló architektúrával rendelkeznek, amely az adott szakterület alapfogalmait tükrözi. Ezek az alkalmazásarchitektúrák meglehetősen általánosak is lehetnek, mint amilyen az információs rendszerek architektúrája, de lehetnek speciálisak is. Például egy termékvonal olyan alkalmazások összességét takarja, amelyek ugyanazon architekturális mag köré épülő változatok segítségével elégíti ki a különféle vásárlói igényeket. Az architektúra megtervezésekor meg kell határozni, hogy rendszerünkben, illetve az egyéb esetleges szakterületi alkalmazásokban mi a közös, és hogyan lehetne ezt a tudást újrafelhasználni. Általános alkalmazásarchitektúrákról a 13., termékvonalakról pedig a 18. fejezetben írok részletesebben.

Beágyazott és személyi számítógépes rendszerek esetében általában csak egy processzorral dolgozunk, és így nincs szükség arra, hogy elosztott architektúrát tervezzünk rendszerünk számára. A legtöbb nagy rendszer azonban manapság olyan elosztott rendszer, ahol a rendszerszoftvert megosztják különböző számítógépek között. Az elosztás architektúrájának helyes megválasztása kulcsfontosságú a rendszer teljesítménye és megbízhatósága szempontjából. Ez fontos téma, ezért a 12. fejezetben külön tárgyalom.

Egy szoftverrendszer architektúrája egy bizonyos architekturális modellen vagy stíluson alapulhat. Az architekturális stílus nem más, mint a rendszer szervezésének egy mintája (Garlan és Shaw, 1993), mint amilyen például a kliens–szerver vagy a rétegzett architektúra. Ezen stílusoknak, alkalmazási korlátaiknak, erősségeiknek és gyenge pontjaiknak az ismerete roppant fontos. A legtöbb nagy rendszer architektúrája azonban nem egyetlen stílus alapján épül fel, a rendszer különböző részeit különböző architekturális stílus segítségével lehet megtervezni. Bizonyos esetekben a teljes rendszer-architektúra összetett lehet, amelyet különböző architekturális stílusok kombinálásával kapunk.

Garlan és Shaw architekturális stílusról alkotott elképzelése lefedi a következő három tervezési kérdést. Ki kell választanunk a követelményeknek kielégítését lehetővé tévő legmegfelelőbb szerkezetet (például kliens–szerver vagy rétegzett). A rendszer egységeinek modulokra bontásához meg kell határoznunk az alrendszerek komponensekre vagy modulokra bontásának stratégiáját. A használható megközelítések különféle architektúrákat tesznek lehetővé. Végül a vezérlés modellezése során azt kell eldöntenünk, hogy hogyan vezéreljük az alrendszerek végrehajtását. Ki kell alakítani a létrejött rendszer részei közötti vezérlőkapcsolatok általános modelljét. Erről a három témáról a 11.2–11.4. alfejezetekben írok.

Egy architekturális tervet nehéz értékelni, mivel az architektúra igazi tesztjét a telepítés után tudjuk elvégezni, vizsgálva a funkcionális és nemfunkcionális követelményeknek történő megfelelést. Bizonyos esetekben azonban értékelhetjük a tervünket referenciamodellekkel vagy általános architekturális modellekkel történő összehasonlítás segítségével. Referenciaarchitektúrákkal a 11.5. alfejezetben, míg egyéb általános architektúrákkal a 13. fejezetben foglalkozom.

Az architekturális tervezés folyamatának eredménye egy architekturális tervezési dokumentum. Ez a rendszer grafikus reprezentációit tartalmazza, a hozzájuk kapcsolódó leíró szöveggel együtt. Tartalmaznia kell annak leírását, hogy a rendszert hogyan lehet alrendszerekre bontani, az egyes alrendszerek miképpen oszthatók modulokra, és ezekhez milyen megközelítést alkalmazunk. A rendszer grafikus modelljei az architektúra különböző nézőpontjait tükrözik. A kialakítandó architekturális modellek többek között az alábbiak lehetnek:

1. A *statikus szerkezeti modell* azt mutatja meg, hogyan lehet az alrendszereket és komponenseket különálló egységként fejleszteni.
2. A *dinamikus folyamatmodell* azt ábrázolja, hogy a rendszer hogyan szervezhető futási idejű folyamatokba. Ez különbözhet a statikus modelltől.
3. Az *interfészmodell* az alrendszerek által publikus interfészeiken keresztül nyújtott szolgáltatásait írja le.
4. A *kapcsolatmodellek* az alrendszerek közötti kapcsolatokat (például adatáramlás) mutatják be.
5. Az *elosztási modell* azt adja meg, hogy az egyes alrendszereket hogyan kell elosztani a számítógépek között.

A rendszer-architektúrák leírására számos kutató az architektúraleíró nyelvek (architectural description languages, ADL) használatát javasolja. Bass és mások (Bass *és mások*, 2003) megadták ezen nyelvek fő jellemzőit. Az ADL-ek alapelemei komponensek és összekötők (konnektorok), valamint szabályokat és irányelveket tartalmaznak a jól formált architektúrák építésére vonatkozóan. A specializált nyelvekhez hasonlóan azonban az ADL-eket csak szakértők értik, a szakterület és az alkalmazás szakemberei nem. Gyakorlati szempontból ez bonyolulttá teszi elemzésüket. Úgy vélem, hogy emiatt ezeket csak kevés alkalmazás esetén lehet jól használni. Az architekturális leíráshoz leginkább továbbra is az olyan informális modelleket és jelölésrendszereket fogják használni, mint amilyen az UML is (Clements *és mások*, 2002).

11.2. A RENDSZER FELÉPÍTÉSE

Már az architekturális tervezési folyamat korai szakaszában döntenünk kell a rendszer teljes szerkezeti modelljéről. Ezt a szervezési módot az alrendszerek közvetlenül is tükrözhetik, de gyakran előfordul, hogy az alrendszer modellje részletesebb a szerkezeti modellnél, így ez nem minden esetben jelent egyszerű megfeleltetést.

Ebben az alfejezetben három széles körben használt szervezési módszert mutatunk be, nevezetesen egy tárolási modellt, egy kliens–szerver modellt és egy absztrakt gép modellt, ahol a rendszert funkcionális rétegekbe szervezzük. Ezek a szervezési stílusok önállóan és együtt is használhatók. Például egy rendszert szervezhetünk megosztott adattár köré úgy, hogy a minél magasabb fokú adatabsztrakció érdekében e köré rétegeket készítünk.

11.2.1. A tárolási modell

A rendszert felépítő alrendszereknek a hatékony együttműködés céljából információt kell cserélniük, amely alapvetően két módon történhet:

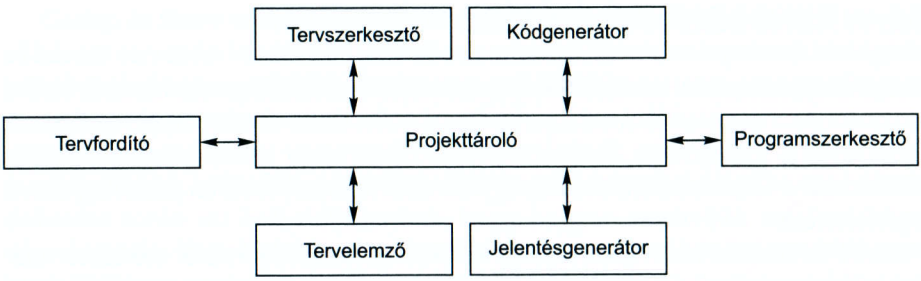
1. Minden megosztott adatot a minden alrendszer által elérhető, központi adatbázisban kell elhelyezni. A megosztott adatbázison alapuló rendszermodellt *tárolási modell*nek nevezzük.
2. Minden alrendszer saját adatbázist tart fent. Ekkor az egyéb alrendszerek közötti adatcsere üzenetküldés segítségével történik.

A nagy mennyiségű adatokkal dolgozó rendszerek osztott adatbázisok és tárolók köré szerveződnek. Ez a modell azon alkalmazások számára megfelelő, ahol az egyik alrendszerben keletkező adatok egy másik alrendszerben kerülnek felhasználásra. Ilyen rendszerek például a parancs- és vezérlőrendszerek, a vezetői információs rendszerek, a CAD-rendszerek és a CASE-eszközkészletek.

A 11.2. ábra egy megosztott tárolóra épülő CASE-eszközkészlet architektúráját mutatja be. Az első megosztott tárolót CASE-eszközkészletek számára valószínűleg az 1970-es évek elején egy ICL nevű, egyesült királyságbeli cég fejlesztette ki, operációs rendszerük fejlesztésének támogatása céljából (McGuffin *és mások*, 1979). A modell akkor vált szélesebb körben ismertté, amikor Buxton (Buxton, 1980) a Stoneman-környezetet javasolta az Ada nyelven írt rendszerek fejlesztésének támogatására. Azóta sok megosztott tárolón alapuló CASE-eszközkészletet fejlesztettek ki.

A megosztott tárolók előnyei és hátrányai a következők:

1. Hatékony módszer nagy mennyiségű adat megosztására, nem szükséges az adatokat explicit módon átvinni egyik alrendszerből a másikba.
2. Az alrendszereknek azonban azonos adattárolási modellel kell rendelkezniük. Ez elkerülhetetlenül az egyes eszközök speciális szükségletei közötti kompro-



11.2. ábra. Integrált CASE-eszközkészlet architektúrája

misszumhoz vezet, ami azonban rossz irányba befolyásolhatja a teljesítményt. A közös sémának nem megfelelő adatmodellel rendelkező új alrendszerek integrálása nehézzé vagy lehetetlenné válik.

3. Az adatokat termelő alrendszereknek foglalkozniuk kell azzal, hogy a többi alrendszer hogyan fogja használni azokat az adatokat.
4. A közös adatmodellnek megfelelően létrejövő információk mennyiségének növekedésével azonban az evolúció egyre bonyolultabbá válhat, ezeknek egy új modellre történő átvitele igen költséges, bonyolult vagy esetleg lehetetlen.
5. Az olyan tevékenységek, mint a biztonsági mentés, a védelem, a hozzáférés-szabályozás és a hiba utáni visszaállítás központosítottak, így a tároló kezelőjének felelősségi körébe tartoznak. Az eszközök lényeges feladataikra összpontosíthatnak anélkül, hogy más kérdésekkel foglalkozniuk kellene.
6. A különböző alrendszerek azonban különböző követelményeket támaszthatnak a védelem, a visszaállítás és a biztonsági mentés lehetőségeit illetően. A tárolási modell ugyanazt a politikát kényszeríti rá minden alrendszerre.
7. A megosztottság modellje a tárolási sémán keresztül látható. Ez egyenes utat biztosít az új eszközök integrálásához, amennyiben azok megfelelnek a közös adatmodellnek.
8. A tároló több gép közötti elosztása azonban bonyolult is lehet. Habár lehetőség van egy logikailag központosított tároló elosztására, problémák léphetnek fel az adatok redundanciájával és inkonzisztenciájával kapcsolatban.

A fenti modellben a tároló passzív, a vezérlés pedig a tárolót használó alrendszerek felelősségi körébe tartozik. A mesterséges intelligencia rendszerek számára alternatív megközelítést dolgoztak ki: ez olyan „modellvázlatot" használ, amely értesíti az alrendszereket, amikor bizonyos adatok elérhetővé válnak. Ez abban az esetben megfelelő, ha a tároló adatai kevésbé jól strukturáltak. Arról, hogy melyik eszközt kell aktiválni, csak az adatok elemzése után lehet dönteni. Ezt a modellt Nii (Nii, 1986) írja le, Bosch (Bosch, 2000) pedig azt tárgyalja, hogy ez a szervezési mód hogyan kapcsolódik a rendszer minőségi jellemzőihez.

11.2.2. A kliens–szerver modell

A kliens–szerver architektúra modellje egy osztott rendszer modellje, amely megmutatja, hogy az adat és a feldolgozás hogyan oszlik meg feldolgozóegységek között. A modell fő komponensei:

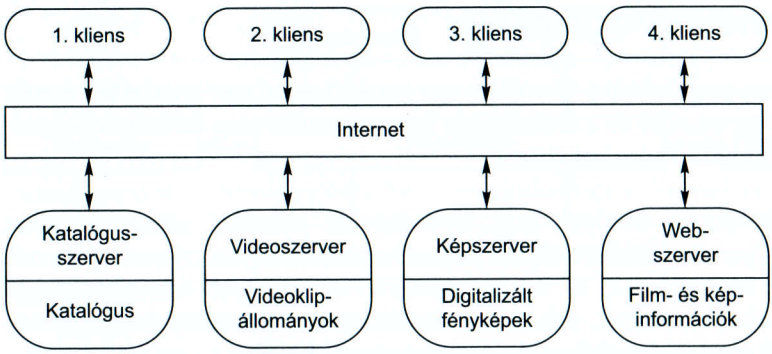
1. Szerverek halmaza, amelyek más alrendszerek számára szolgáltatásokat nyújtanak. Ilyenek például a nyomtatószerverek, amelyek nyomtatási szolgáltatásokat biztosítanak, az állományszerverek, amelyek állománykezelő szolgáltatásokat ajánlanak, valamint a fordítószerverek, amelyek programozási nyelvről történő fordítói szolgáltatásokat nyújtanak.
2. Kliensek halmaza, amelyek hozzáférnek a szerverek által biztosított szolgáltatásokhoz. Ezek általában önálló létjogosultsággal rendelkező alrendszerek. Egy kliensprogramnak számos példánya futhat egyidejűleg.
3. Hálózat, amely lehetővé teszi, hogy a kliensek hozzáférjenek a szolgáltatásokhoz. Ez nem szükséges akkor, ha mind a kliensek, mind pedig a szerverek egyetlen gépen futnak, azonban a gyakorlatban a legtöbb kliens–szerver rendszert elosztott rendszerként valósítják meg.

A klienseknek szükségük lehet arra, hogy ismerjék az elérhető szerverek és az általuk biztosított szolgáltatások neveit, de a szervereknek nem kell tudniuk sem a kliens azonosságát, sem pedig azt, hogy hány kliens van. A kliensek a szerverek által biztosított szolgáltatásokat távoli eljáráshívásokkal érik el, egy kérés-válasz alapú protokoll segítségével, mint például a www esetén használt http protokoll. Lényegében a kliens elküld egy kérést a szervernek, és addig vár, amíg választ nem kap.

A 11.3. ábrán egy kliens–szerver modellre épült rendszer látható. Ez egy többfelhasználós, webalapú könyvtári rendszer filmek és képek kezelésére, amelyben számos szerver kezeli és jeleníti meg a különböző típusú médiumokat. A filmkockákat gyorsan és egy időben, de viszonylag alacsony felbontás mellett kell átvinni. Ezek tárolhatók tömörítve is, azaz a videoszerver több formátum közötti videotömörítést is végez. Az állóképeket azonban nagy felbontással kell elküldeni, ezért célszerű azokat külön szerveren tartani.

A katalógusnak sokféle lekérdezést kell tudnia kezelni és hiperhivatkozásokat kell biztosítania ahhoz a webes információs rendszerhez, amely a film vagy videoklip adatait tartalmazza, és ahhoz az e-üzleti rendszerhez, amelynek segítségével az adott film vagy klip megvásárolható. A kliensprogram egyszerűen egy integrált webes felhasználói felület ezekhez a szolgáltatásokhoz.

A kliens–szerver modell legfontosabb előnye osztott architektúrája. Hatékonyan használható sok, osztott feldolgozási egységből álló hálózati rendszerek esetén. Egy új szerver könnyen hozzáadható a rendszerhez és integrálható annak többi részével. A szerverek könnyen, átlátszó módon felújíthatók anélkül, hogy az a rendszer más részeire befolyással lenne. Az osztott architektúrákról – beleértve a kliens–szerver architektúrákat és az osztott objektumarchitektúrákat is – részletesebben a 12. fejezetben szólok.



11.3. ábra. Egy film és kép könyvtári rendszer architektúrája

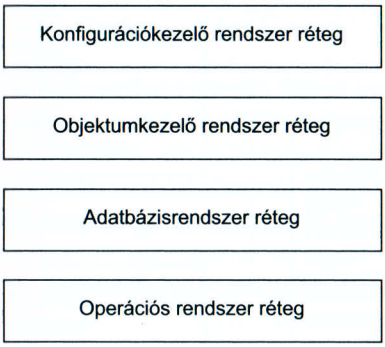
Az új szerverek integrálása által biztosított előnyök megszerzéséhez azonban szükség lehet a már létező kliensek és szerverek változtatására. Lehet, hogy nem létezik a szerverek között megosztott adatmodell, és az egyes alrendszerek is különbözőképpen szervezhetik adataikat. Ez azt jelenti, hogy az egyes szerverek számára a teljesítmény optimalizálását lehetővé tévő specifikus adatmodellek hozhatók létre. Természetesen, ha az adatokat XML-alapon ábrázoljuk, az egyik sémából másikba történő konverzió viszonylag egyszerűvé válik. Az XML azonban az adatábrázolásnak nem túl hatékony módja, így ezt alkalmazva teljesítményproblémákba ütközhetünk.

11.2.3. A rétegzett modell

Egy architektúra rétegzett modellje (amit néha absztrakt gép modellnek is neveznek) a rendszert rétegekbe szervezi, amelyek mindegyike valamilyen szolgáltatásokat biztosít. Minden ilyen rétegre úgy tekinthetünk, mint egy *absztrakt gép*re, amelynek gépi nyelvét a réteg által biztosított szolgáltatások definiálják. Ezt a „nyelvet" használják az absztrakt gép következő szintjének megvalósítására. Például egy nyelv implementálásának szokásos módja az, hogy definiálni kell egy olyan képzeletbeli „nyelvgépet", amelynek kódjára a nyelvet lefordítják. Egy további fordítással ezt az absztrakt gépi kódot valós gépi kóddá kell alakítani.

A rétegzett modellre példa a hálózati protokollok OSI-referenciamodellje (Zimmermann, 1980), amelyről a 11.5. alfejezetben lesz szó. Buxton (1980) egy másik jó példát mutatott be az Ada Programming Support Environment (APSE) háromrétegű modelljével. A 11.4. ábra az APSE szerkezetét tükrözi, és azt mutatja meg, hogy egy konfigurációkezelő rendszer hogyan integrálható az absztrakt gép modell segítségével.

A konfigurációkezelő rendszer objektumok verzióinak kezelését végzi, valamint a 29. fejezetben leírtak szerinti általános konfigurációkezelési lehetőségeket biztosít. Ennek megsegítésére egy objektumkezelő rendszert használ, amely az objektumok számára információtárolási és -kezelési szolgáltatásokat biztosít. Ez



11.4. ábra. Egy verziókezelő rendszer rétegzett modellje

utóbbi rendszer egy adatbázisrendszer segítségével biztosítja az adatok tárolását és az olyan alapvető szolgáltatásokat, mint a tranzakciókezelés, a visszagörgetés, a visszaállítás és a hozzáférés-szabályozás. Az adatbázis-kezelő implementációjában az alatta elhelyezkedő operációs rendszer képességeit és állománytárolási mechanizmusát használja. Egyéb rétegezett architektúrájú modellekre a 13. fejezetben olvashatunk példákat.

A rétegalapú megközelítés segíti a rendszerek inkrementális fejlesztését. Mihelyt egy réteg kifejlesztésre került, a réteg által biztosított szolgáltatások egy része elérhetővé tehető a felhasználók számára. Ez az architektúra ezenfelül változtatható és hordozható is. A réteg helyettesíthető egy másik, azzal ekvivalens réteggel, ha interfésze nem változik meg. Sőt egy réteg interfészének megváltozása vagy a réteg új lehetőségekkel történő bővítése csak a szomszédos réteget érinti. Mivel a rétegzett rendszerek a gépfüggőséget a belső rétegekbe zárják be, ez leegyszerűsíti a többplatformos alkalmazások készítését. Csak a belső, gépfüggő rétegeket kell újraimplementálni ahhoz, hogy figyelembe vegyük egy másik operációs rendszer vagy adatbázis lehetőségeit.

A rétegzett megközelítés hátránya, hogy ily módon a rendszerek strukturálása igen bonyolulttá válhat. Az alapvető adottságokat – mint például az állománykezelést, amelyre minden absztrakt gépnek szüksége van – a belső rétegek biztosíthatják. A külső réteg felhasználó által igényelt szolgáltatásainak „át kell ütniük" több szomszédos réteget is ahhoz, hogy hozzáférjenek a több réteggel alattuk elhelyezkedő réteg szolgáltatásaihoz. Ez felforgatja a modellt abban az értelemben, hogy egy külső réteg a továbbiakban már nem csupán egyszerűen a közvetlen megelőzőjétől függ.

A teljesítmény kérdése szintén problémás a parancsértelmezés néha szükségszerűen több szintje miatt. Amennyiben sok réteg van, egy legfelső rétegből érkező szolgáltatáskérést különféle rétegekben többször is értelmezni kell annak feldolgozása előtt. Hogy elkerüljük ezeket a problémákat, az alkalmazásoknak közvetlenül a belső rétegekkel kell kommunikálniuk ahelyett, hogy a szomszédos réteg által biztosított szolgáltatásokat használnák.

11.3. MODULÁRIS FELBONTÁS

Miután a teljes rendszer-architektúra kiválasztásra került, dönteni kell az alrendszerek modulokra bontása során alkalmazott megközelítésről. Nincs szigorú különbség a rendszer szervezése és a moduláris felbontás között. A 11.2. alfejezetben leírt stílusok ezen a szinten is alkalmazhatók. A modulokban található komponensek azonban általában kisebbek, mint az alrendszerek, így alternatív felbontási stílusok használata is lehetővé válik.

Az alrendszerek és modulok nem különböztethetők meg egyértelműen, de érdemes azokat a következő módon elképzelni:

1. Egy alrendszer egy olyan önálló rendszer, amely működése nem függ más alrendszerek szolgáltatásaitól. Az alrendszerek modulokból épülnek fel, és az egyéb alrendszerekkel interfészeken keresztül kommunikálnak.
2. Egy modul olyan rendszerkomponens, amely más modulok számára szolgáltatás(oka)t biztosít. Általában nem tekintjük önálló rendszernek. A modulok rendszerint egyszerűbb rendszerkomponensekből épülnek fel.

Az alrendszerek modulokra bontása során két fő stratégia ismeretes:

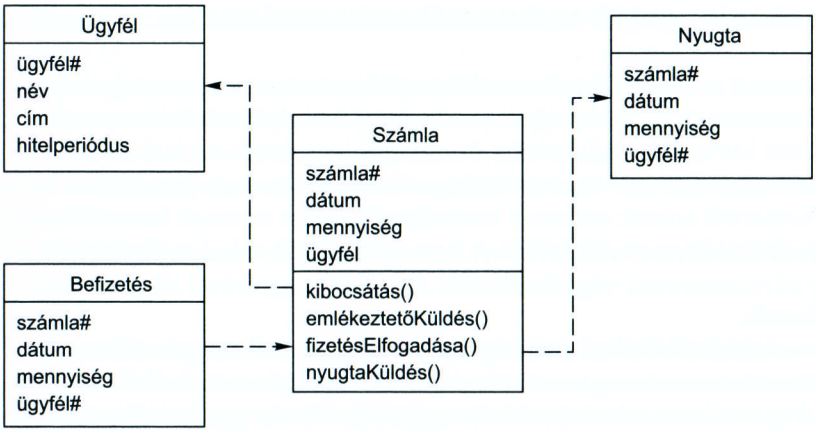
1. *Objektumorientált felbontás*sal a rendszert egymással kommunikáló objektumok halmazára bontjuk fel.
2. *Funkcióorientált csővezetékek* használatával a rendszert bemenő adatokat elfogadó és azokat kimenő adatokká alakító funkcionális modulokra bontjuk.

Az objektumorientált megközelítésben a modulok egyéni állapottal, valamint az ezeken az állapotokon értelmezett műveletekkel rendelkező objektumok. A csővezetékelvű modellben a modulok funkcionális transzformációk. A modulok mindkét esetben szekvenciális komponensként vagy folyamatként valósíthatók meg.

Lehetőség szerint nem szabad elhamarkodottan döntenünk a rendszer konkurenciájáról. A konkurens rendszertervezés elkerülésének előnye, hogy a szekvenciális programok könnyebben tervezhetők, implementálhatók, ellenőrizhetők és tesztelhetők, mint a párhuzamos rendszerek. A folyamatok közötti időbeli függőség nehezen formalizálható, vezérelhető és ellenőrizhető. A legjobb megoldás az, hogy először modulokra kell bontani a rendszereket, majd az implementáció során kell eldönteni, hogy azokat szekvenciálisan vagy párhuzamosan kell-e végrehajtani.

11.3.1. Objektumorientált felbontás

Egy objektumorientált architekturális modell a rendszert lazán kapcsolódó, jól definiált interfészekkel rendelkező objektumok halmazára tagolja. Az objektumok a többi objektum által biztosított szolgáltatásokat hívják. Az objektummodelleket már a 8. fejezetben bemutattam, részletesen pedig a 14. fejezetben tárgyalom.



11.5. ábra. Egy számlafeldolgozó rendszer objektummodellje

A 11.5. ábrán egy számlafeldolgozó rendszer objektumorientált architekturális modelljének példája látható. A rendszer számlákat bocsáthat ki az ügyfelek számára, befizetéseket vehet át, a befizetéseket igazoló nyugtákat, valamint a kifizetetlen számlákról fizetési felszólításokat adhat ki. A 8. fejezetben bemutatott UML jelölésrendszerét használom, ahol az objektumosztályok névvel és attribútumokkal rendelkeznek. A műveletek – ha vannak – az objektumot reprezentáló téglalap alsó részében helyezkednek el. A szaggatott nyilak azt jelölik, hogy az objektum egy másik objektum attribútumait vagy az általa biztosított szolgáltatásokat használja.

Az objektumorientált felbontás objektumosztályokkal, azok attribútumaival és műveleteivel foglalkozik. Implementációkor az objektumok ezekből az osztályokból jönnek létre, és az objektum műveleteinek koordinálásához valamilyen vezérlési modellt alkalmaznak. Ebben a példában a Számla osztály rendelkezik a rendszer funkcionalitását megvalósító különféle műveletekkel. Ez az osztály használja az ügyfeleket, befizetéseket és nyugtákat reprezentáló osztályokat.

Az objektumorientált megközelítés előnyei jól ismertek. Mivel az objektumok lazán kapcsolódnak, az objektumok implementációja változtatható anélkül, hogy az hatással lenne más objektumokra. Az objektumok gyakran a valós világ egyedeinek reprezentációi, így a rendszer szerkezete könnyen megérthető. Mivel a valós világ egyedei különböző rendszerekben is létezhetnek, az objektumok újrafelhasználhatók. Az architekturális komponensek közvetlen implementációjának biztosítására objektumorientált programozási nyelveket fejlesztettek ki.

Azonban az objektumorientált megközelítésnek vannak hátrányai is. A szolgáltatások használata érdekében az objektumoknak explicit módon kell hivatkoznia a többi objektum nevére és interfészére. Ha a rendszer tervezett változtatásához az interfész megváltoztatására van szükség, akkor a változtatást minden, a megváltozott objektumot használó helyen át kell vezetni. Míg az objektumok tisztán leképezhetők egyszerű valós világbeli egyedekre, az összetettebb egyedeket gyakran nehezen lehet objektumként reprezentálni.

11.3.2. Funkcióorientált csővezetékek használata

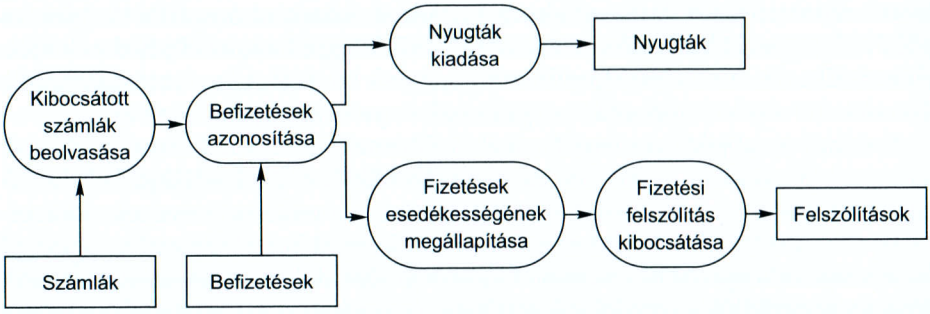
A funkcióorientált csővezetékek használata esetén, amit más néven adatfolyammodellnek is neveznek, funkcionális transzformációk dolgozzák fel bemenetüket és hoznak létre kimeneteket, közöttük áramlanak az adatok és sorban előrehaladva kerülnek átalakításra. Minden feldolgozási lépés transzformációként valósul meg. A bemeneti adatok ezeken a transzformációkon mennek keresztül, míg végül átalakulnak kimeneti adatokká. A transzformációk mind szekvenciálisan, mind pedig párhuzamosan végrehajthatók. Az adatok egyesével vagy kötegelve is feldolgozhatók.

Ha a transzformációk különböző folyamatokként vannak reprezentálva, akkor ezt a modellt a Unix-rendszer terminológiáját követve cső- és szűrőstílusnak nevezzük. A Unix-rendszer adatvezetékként működő csöveket, valamint parancsokat biztosít, amelyek funkcionális transzformációk. Az ilyen modellnek megfelelő rendszerek a csöveket használó Unix-parancsok és a Unix-héj vezérlési lehetőségeinek kombinálásával valósíthatók meg. A *szűrő* szót azért használjuk, mert a transzformáció „kiszűri" a bejövő adatáramból az általa feldolgozható adatokat.

A csővezetékelvű modell különböző változatait azóta alkalmazzák, mióta a számítógépeket először használták automatikus adatfeldolgozásra. Ha a transzformáció szekvenciális, az adatfeldolgozás pedig kötegelt, akkor ezt az architekturális modellt kötegelt szekvenciális modellnek nevezzük. Ahogyan arról a 13. fejezetben írok, ez az olyan adatfeldolgozó rendszerek szokásos architektúrája, mint amilyen például egy számlázórendszer. Az adatfeldolgozó rendszerek általában nagyszámú bemeneti rekordból egyszerű számításokkal sok kimeneti jelentést hoznak létre.

Egy ilyen rendszer architektúrája látható a 11.6. ábrán. Egy szervezet számlákat bocsát ki ügyfelei számára. Hetente egyszer összevetik az addig történt befizetéseket a számlákkal. A kifizetett számlákról nyugtát bocsát ki, míg a fizetési határidőn belül ki nem fizetett számlákról fizetési felszólítást küld.

Ez a modell azonban csupán része a számlafeldolgozó rendszernek: a számlák kibocsátására alternatív transzformációk is használhatók. Figyeljük meg a különbséget e között és az előző szakaszban leírt objektumorientált megfelelője között! Az objektummodell absztraktabb, mivel nem hordoz információt a műveletek sorrendjéről.



11.6. ábra. Egy számlafeldolgozó rendszer adatfolyammodellje

Ennek az architektúrának a következő előnyei vannak:

1. Támogatja a transzformációk újrafelhasználhatóságát.
2. Könnyen érthető abban az értelemben, hogy sok ember a saját munkáját a bemenetek és kimenetek feldolgozásaként értelmezi.
3. A rendszer új transzformációk hozzáadásával történő bővítése általában egyszerű.
4. Mind konkurens, mind pedig szekvenciális környezetben könnyen implementálható.

A modell alapvető hátránya az összes transzformáció által felismerhető közös adatátviteli formátum igényéből származik. Az egyes transzformációknak vagy egyeztetniük kell egymással az átadandó adatok formátumát, vagy a kommunikációban részt vevő adatok számára egy szabványos formátumot kell kialakítani. Az utóbbi csak önálló és újrafelhasználható transzformációk esetén használható. A Unixban a szabványos formátum egyszerűen egy karaktersorozat. Minden transzformáció a megegyezés szerinti formátumban kapja a bemeneti adatokat, és ugyanabban kell produkálnia a kimenő adatokat is. Ez azonban a rendszer túlterhelését növeli, és azt is jelentheti, hogy az inkompatibilis adatformátumokat használó transzformációk integrálása lehetetlenné válik.

Az interaktív rendszereket nehéz csővezetékelvű modell alapján megírni a feldolgozandó adatáram iránti szükséglet miatt. Míg az egyszerű szöveges be-, illetve kimenet modellezhető ily módon, a grafikus felhasználói felületek sokkal összetettebb I/O-formátumokat és olyan, eseményeken alapuló vezérlést tartalmaznak, mint amilyen az egérkattintás és a menükiválasztás. Ezt nehéz a csővezetékelvű modellel kompatibilis formára hozni.

11.4. VEZÉRLÉSI STÍLUSOK

Egy rendszer strukturális modelljei azzal foglalkoznak, hogy a rendszer hogyan bontható fel alrendszerekre. Ahhoz, hogy az alrendszerek rendszerként működjenek, úgy kell őket vezérelni, hogy szolgáltatásaik a megfelelő helyre a megfelelő időben eljussanak. A strukturális modellek nem tartalmaznak (és nem is szabad tartalmazniuk) vezérlési információkat. Ehelyett a rendszer kiépítőjének az alrendszereket az alkalmazott strukturális modellt kiegészítő valamilyen vezérlési modellnek megfelelően kell szerveznie. Az architekturális szinten elhelyezkedő vezérlési modellek az alrendszerek közötti vezérlési folyamatokkal foglalkoznak.

A szoftverrendszerekben két általános vezérlési stílust alkalmaznak:

1. *Központosított vezérlés.* A vezérlés teljes felelősségét egyetlen alrendszer látja el, amely beindítja és leállítja a többi alrendszert. A vezérlést át is háríthatja egy másik alrendszerre, de elvárás, hogy a vezérlés felelőssége visszatérjen hozzá.

2. *Eseményalapú vezérlés.* Ahelyett, hogy a vezérlési információ egyetlen alrendszerbe lenne beágyazva, minden alrendszer válaszolhat egy külsőleg létrejött eseményre. Ezek az események vagy más alrendszerekből, vagy a rendszer környezetéből származhatnak.

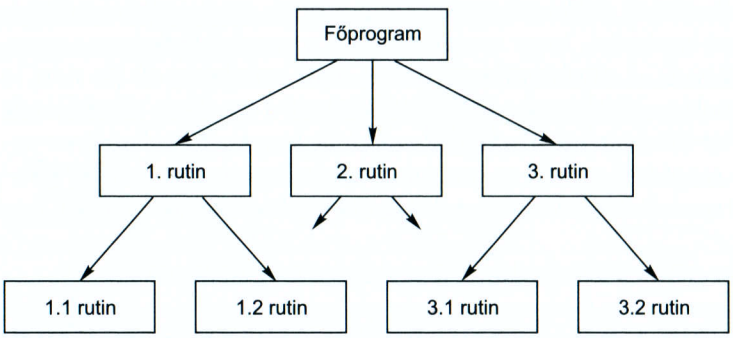
A vezérlési stílusok kiegészítik a strukturális stílusokat. Minden korábban bemutatott strukturális stílust meg lehet valósítani központosított és eseményalapú vezérléssel egyaránt.

11.4.1. Központosított vezérlés

A központosított vezérlési modellben van egy kitüntetett alrendszer, a rendszervezérlő, amely a többi alrendszer végrehajtásáért felelős. A központosított vezérlési modellek két csoportba oszthatók attól függően, hogy a vezérelt alrendszerek szekvenciálisan vagy párhuzamosan hajtódnak-e végre.

1. *A hívás-visszatérés modell.* Ez a megszokott fentről le alprogram modellje, ahol a vezérlés az alprogram-hierarchia csúcsán kezdődik és alprogramhívások segítségével jut el a fa alsóbb szintjeire. Ez a modell csak szekvenciális rendszerek esetén alkalmazható.
2. *A kezelőmodell.* Konkurens rendszerekre alkalmazható. Ebben egy kijelölt rendszerkezelő rendszerkomponens irányítja a többi rendszerfolyamat indítását, leállítását, valamint koordinálja azokat. A folyamat olyan alrendszer vagy modul, amely végrehajtható más folyamatokkal párhuzamosan. Ez a fajta modell használható szekvenciális rendszerek esetén is, ahol egy kezelőrutin, állapotváltozók értékeitől függően, meghív egyes alrendszereket. Ezt általában többirányú elágaztató utasítással valósítják meg.

A hívás-visszatérés modell a 11.7. ábrán látható. A főprogram az 1-es, 2-es és 3-as rutinokat hívhatja; az 1-es rutin az 1.1-es és 1.2-es; a 3-as pedig a 3.1-es és 3.2-



11.7. ábra. A vezérlés hívás-visszatérés modellje

es rutinokat hívhatja stb. Ez a program dinamikájának, *nem* pedig struktúrájának modellje: nem szükséges, hogy például az 1.1-es rutin az 1-es része legyen.

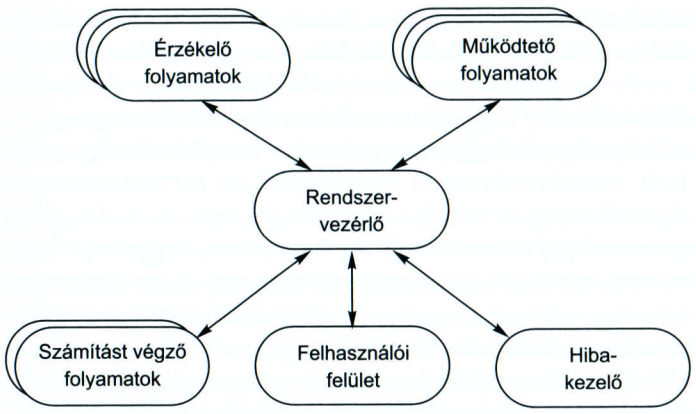
Ez az ismert modell van beágyazva az olyan programozási nyelvekbe, mint például a C, az Ada vagy a Pascal. A vezérlés a hierarchiában magasabb szintű rutintól jut el az alacsonyabb szinten lévőhöz, majd visszatér a hívás helyére. A vezérlés felelőssége a végrehajtás alatt álló alprogramoké, és meghívhatnak egyéb rutinokat, vagy visszaadhatják a vezérlést a szülőnek. A program egyéb pontjára történő visszatérés igénytelen programozási stílusra vall.

A hívás-visszatérés modell modulszinten használható a tevékenységek és objektumok vezérlésére. Sok objektumorientált rendszerben az objektumok műveletei (metódusok) eljárásként vagy függvényként vannak implementálva. Például amikor egy Java-objektum igénybe szeretné venni egy másik objektum szolgáltatásait, azt a megfelelő módszer meghívásával teszi.

A modell merev és korlátozott természete egyben előny és hátrány is. Erőssége a vezérlési folyam elemzésének és bizonyos bemenet esetén a rendszer válasza kiszámíthatóságának viszonylagos egyszerűsége. Hátránya, hogy a normálistól eltérő működés esetén használata kényelmetlen, mint ahogyan azt majd a 18. fejezetben kifejtem.

A 11.8. ábra egy konkurens rendszer vezérlésének központosított kezelési modelljét ábrázolja. Ezt a modellt gyakran használják olyan „gyengén" valós idejű rendszerekben, ahol nincsenek nagyon szűk időkorlátok. A központi vezérlő irányítja az érzékelőkkel és működtetőkkel kapcsolatban álló folyamatok végrehajtását. A 15. fejezetben leírt épületfigyelő rendszer ezt a vezérlési modellt alkalmazza.

A rendszert vezérlő folyamat a rendszer állapotváltozói alapján dönti el, hogy az egyes folyamatokat mikor kell elindítani, illetve leállítani. Ellenőrzi, hogy a többi folyamat hozott-e létre feldolgozandó vagy feldolgozásra továbbküldendő információt. A vezérlő általában ciklikusan ellenőrzi az érzékelőket és folyamatokat, hogy bekövetkezett-e valamilyen esemény vagy állapotváltozás. Éppen ezért ezt a modellt eseményciklus-modellnek is szokás nevezni.



11.8. ábra. Valós idejű rendszerek központosított vezérlési modellje

11.4.2. Eseményvezérelt rendszerek

Központosított vezérlési modellekben a vezérlési döntéseket általában a rendszer néhány állapotváltozójának az értéke határozza meg. Ezzel szemben az eseményvezérelt vezérlési modelleket külsőleg létrehozott események irányítják. Ebben a környezetben az *esemény* fogalma alatt nem csupán egy bináris jelet kell érteni, hanem lehet egy értéktartományt befutó jel vagy egy menüből kapott parancs is. Egy esemény és egy egyszerű bemenet között az a különbség, hogy az esemény időzítése az őt kezelő folyamat fennhatóságán kívül helyezkedik el.

Az eseményvezérelt rendszereknek sok fajtája ismeretes, beleértve a szerkesztőket, ahol a szerkesztőutasításokat felhasználói felület események jelzik; az olyan szabályalapú termelő rendszereket, mint amilyeneket a mesterséges intelligenciában használnak, ahol egy feltétel igazzá válása kivált valamilyen tevékenységet; az aktív objektumokat, ahol egy objektum attribútumértékének megváltozása idéz elő valamilyen tevékenységet. A különböző ilyen típusú rendszerekről írnak Garlan és mások (Garlan *és mások*, 1992).

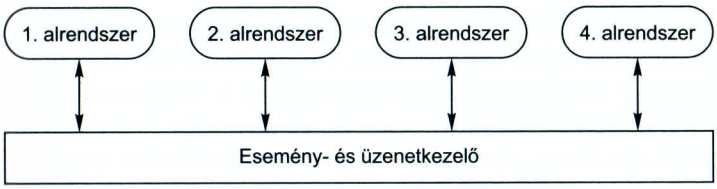
Ebben a szakaszban két eseményvezérelt vezérlési modellt mutatunk be:

1. *Eseményszóró modellek.* Ezekben a modellekben egy esemény minden alrendszerhez eljut, és bármelyik, az esemény kezelésére programozott alrendszer reagálhat rá.
2. *Megszakításvezérelt modellek.* Ezeket kizárólag olyan valós idejű rendszerekben használják, ahol egy megszakításkezelő észleli a külső megszakításokat. Ezek aztán valamely más komponenshez kerülnek feldolgozásra.

Az eseményszóró modellek igen hatékonyan integrálják az egy hálózat különböző számítógépei között osztott alrendszereket. A megszakításvezérelt modelleket szigorú ütemezési követelményekkel rendelkező valós idejű rendszerekben használják.

Egy eseményszóró modellben (11.9. ábra) az alrendszerek regisztrálják bizonyos eseményekre vonatkozó érdeklődésüket. Amikor ezek az események bekövetkeznek, a vezérlés átkerül az alrendszerhez, amely kezelheti az eseményt. Az eseményszóró modell és a 11.8. ábrán is látható központosított modell közötti különbség az, hogy a vezérlési politika nincs beépítve az esemény- és üzenetkezelőbe. Az alrendszerek eldöntik, hogy mely eseményekre tartanak igényt, az esemény- és üzenetkezelő pedig biztosítja, hogy ezek az események eljussanak hozzájuk.

Minden esemény eljuttatható akár minden alrendszernek is, de ez rengeteg feldolgozási többletterheléssel jár. Az esemény- és üzenetkezelő gyakran nyilvántartást vezet az alrendszerekről és az őket érdeklő eseményekről. Az alrendszerek a feldolgozás számára elérhető adatokat jelző eseményeket generálnak. Az eseménykezelő érzékeli ezeket az eseményeket, megnézi a nyilvántartást, és továbbítja az eseményeket azoknak az alrendszereknek, amelyek érdeklődésüket fejezték ki az adott esemény iránt. Egyszerűbb rendszerekben – mint amilyenek például a felhasználó felület események által vezérelt PC-alapú rendszerek – kü-



11.9. ábra. Szelektív eseményszóráson alapuló vezérlési modell

lön eseményfigyelő alrendszerek léteznek, amelyek az egértől, billentyűzettől stb. érkező eseményeket figyelik, és azokat speciális utasításokra fordítják.

Az eseménykezelő általában a pont-pont kommunikációt támogatja. Egy alrendszer explicit módon küldhet üzenetet egy másik alrendszernek. Ennek a modellnek számos változata létezik, mint például a Field-környezet (Reiss, 1990) vagy a Hewlett–Packard Softbench (Fromme és Walker, 1993). Mindkettőt az eszközök együttműködésének szoftvertervezési környezetekben történő vezérlésére használják. A 12. fejezetben tárgyalásra kerülő ORB-k szintén támogatják ezt a modellt.

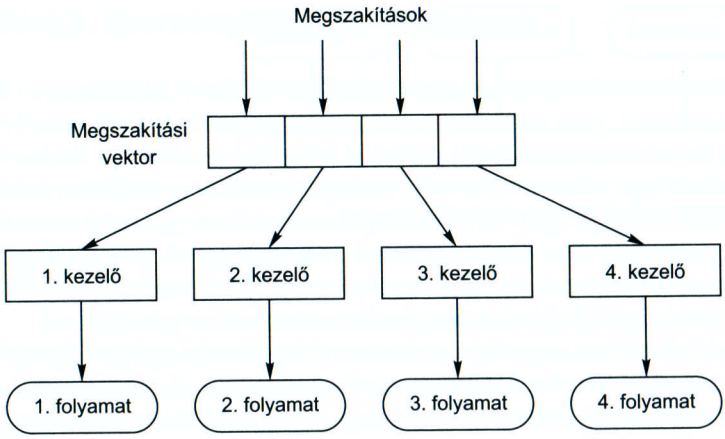
Ennek az eseményszóró megközelítésnek előnye, hogy evolúciója viszonylag egyszerű. Események bizonyos osztályának kezelését végző új alrendszerek eseményeik eseménykezelőnél történő regisztrálásával integrálhatók. Bármely alrendszer aktiválhat bármely egyéb alrendszert anélkül, hogy tudná annak nevét vagy helyét. Az alrendszerek osztott gépeken implementálhatók. Ez az osztottság a többi alrendszerre nézve átlátszó.

A modell hátránya, hogy az alrendszerek nem tudják, hogy az események kezelésre kerülnek-e, és ha igen, mikor. Amikor egy alrendszer létrehoz egy eseményt, nem tudja, hogy mely alrendszerek jelezték érdeklődésüket az adott eseményre. Lehetséges, hogy különböző alrendszerek ugyanarra az eseményre regisztrálták magukat. Ez konfliktushoz vezethet, amikor az eseménykezelés eredményei hozzáférhetővé válnak.

A külsőleg létrejött események nagyon gyors kezelését igénylő valós idejű rendszereknek eseményvezéreltnek kell lenniük. Például egy autó biztonsági rendszereit vezérlő valós idejű rendszernek fel kell ismernie egy ütközést, és fel kell fújnia a légzsákot, mielőtt a sofőr feje a kormánykeréknek ütközik. Ahhoz, hogy biztosítva legyen az eseményekre történő ilyen gyors válaszadás, megszakításalapú vezérlést kell alkalmazni.

A megszakításalapú vezérlés modellje a 11.10. ábrán látható. Adott ismert számú megszakítástípus, valamint mindegyik típus számára egy kezelő. Minden megszakítástípus rendelkezik egy memóriaterülettel, ahol kezelőjének címe helyezkedik el. Amikor egy bizonyos típusú megszakítás érkezik, egy hardverkapcsoló a vezérlést azonnal a kezelőhöz továbbítja. Ez a megszakításkezelő ezután a megszakítással jelzett eseményre adandó válasz gyanánt egyéb folyamatokat indíthat el, illetve állíthat le.

Ez a modell csak olyan bonyolult valós idejű rendszerekben használható, ahol néhány eseményre azonnali válasz szükséges. Kombinálható is a központosított kezelő modellel: a központi kezelő a rendszer normál működését kezeli, míg váratlan események esetén a megszakításalapú vezérlés alkalmazható.



11.10. ábra. Megszakításalapú vezérlési modell

A fenti megközelítés előnye, hogy nagyon gyors válaszadást tesz lehetővé a bekövetkezett eseményekre. Hátránya, hogy bonyolultan programozható és nehezen ellenőrizhető. A rendszer tesztelése során gyakorlatilag lehetetlen megismételni a megszakítás ütemezésének mintáit. Az ezen modellel kifejlesztett rendszerek megváltoztatása igen bonyolult, ha a hardver korlátozza a megszakítások számát. Ennek a határnak az elérése esetén más típusú eseményt nem lehet kezelni. Ez a megszorítás néha kijátszható több eseménytípus egyetlen megszakításra történő leképezésével, de ekkor a kezelőnek kell megfejtenie, hogy mely esemény következett be. Azonban ez a módszer nem célravezető abban az esetben, amikor a megszakításra nagyon gyors választ kell biztosítani.

11.5. REFERENCIAARCHITEKTÚRÁK

A fentebb tárgyalt architekturális modellek általános modellek, különféle alkalmazások esetén alkalmazhatók. Ezen általános modellek mellett használhatók bizonyos szakterületi alkalmazásokra jellemző architekturális modellek is. Habár ezeknek a rendszereknek a példányai részleteiben különböznek, a közös architekturális szerkezet új rendszerek fejlesztése esetén újrafelhasználható. Ezeket az architekturális modelleket *szakterület-specifikus architektúrák*nak nevezzük.

A szakterület-specifikus architekturális modelleknek két típusa van:

1. *Általános modellek,* amelyek számos valós rendszer absztrakciói, ezen rendszerek fő jellemzőit foglalják magukban. Például valós idejű rendszerekben a különböző rendszertípusok – adatgyűjtő rendszerek, figyelő- (monitorozó-) rendszerek stb. – rendelkezhetnek általános architekturális modellekkel. Az alkalmazásarchitektúrákról szóló 13. fejezetben több általános modellt is tárgyalok, ebben az alfejezetben csak a referenciamodellekre összpontosítok.

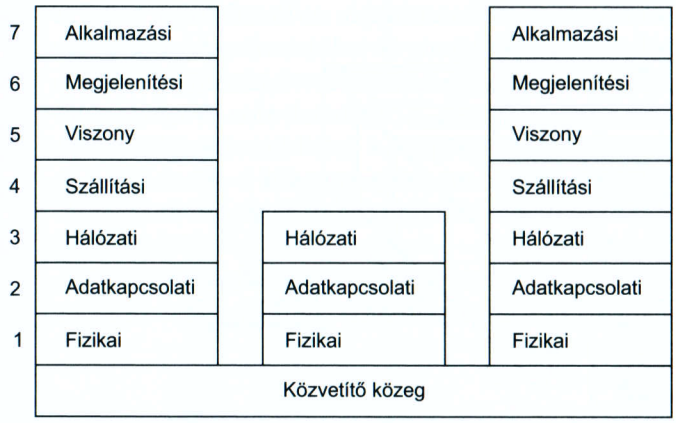
2. *Referenciamodellek,* amelyek absztraktabbak és a rendszerek nagyobb osztályát írják le. Ők informálják a tervezőket az adott rendszer osztályának általános szerkezetéről. A referenciamodellek általában a szakterület valamely tanulmányából származnak, és egy olyan idealizált architektúrát mutatnak be, amely minden olyan jellemzőt tartalmaz, amit a rendszer magában foglalhat.

Természetesen a különféle modellek közötti különbség nem túl merev. Az általános modellek sokszor referenciamodellként is szolgálnak. Azért tettem közöttük különbséget, mert az általános modellek a tervezés során közvetlenül újrafelhasználhatók. A referenciamodelleket normál körülmények között a szakterületi fogalmak közlésére és a lehetséges architektúrák összehasonlítására használják.

Normális esetben a referenciamodellek nem biztosítanak járható utat az implementációhoz. Ehelyett fő feladatuk, hogy a szakterületen belüli különböző rendszerek összehasonlításának eszközéül szolgáljanak. Egy referenciamodell az összehasonlítás szótárát biztosítja, olyan szabványként viselkedik, amely alapján elvégezhető a rendszerek értékelése.

Az OSI-modell a nyílt rendszerek összekapcsolásának a 11.11. ábrán látható hétrétegű modellje. A különböző rétegek pontos feladatai itt nem lényegesek. Röviden, az alsó rétegek a fizikai összekapcsolással, a középső rétegek az adatátvitellel, míg a felső rétegek a jelentéssel bíró alkalmazásinformációk (mint például a szabványosított dokumentumok) átvitelével foglalkoznak.

Az OSI-modell tervezőinek szemei előtt az a nagyon gyakorlatias cél lebegett, hogy olyan szabványt alkossanak meg, amely szabványnak megfelelő rendszerek tudnak egymással kommunikálni. Minden réteg csak az alatta elhelyezkedő rétegtől függhet. A technológia fejlődésével egy réteg átlátszó módon újraimplementálható anélkül, hogy az az egyéb rétegeket használó rendszereket befolyásolná.



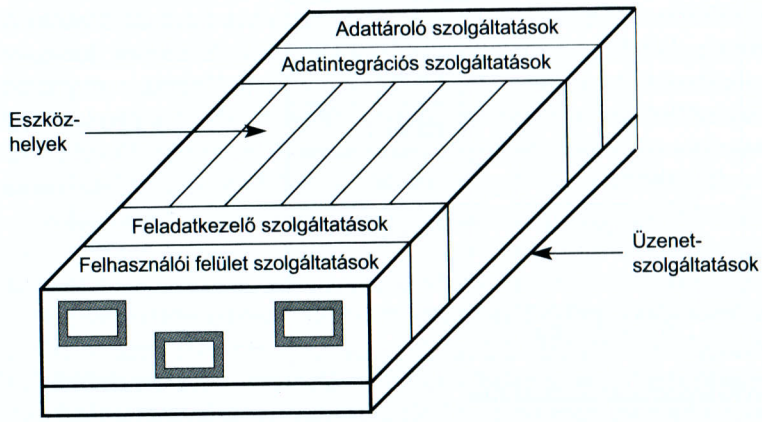
11.11. ábra. Az OSI-referenciamodell architektúrája

A gyakorlatban azonban az architekturális modellezés rétegezett megközelítésének problémái veszélyeztették ezeket a célokat. A hálózatok közötti hatalmas különbségek miatt elképzelhető, hogy az egyszerű összekapcsolás lehetetlen. Habár a rétegek funkcionális jellemzői jól definiáltak, a nemfunkcionális jellemzők nincsenek meghatározva. A rendszerfejlesztőknek implementálniuk kell saját magasabb szintű szolgáltatásaikat, így rétegeket hagynak ki a modellből.

Ebből következően a réteg átlátszó módon történő lecserélése szinte sohasem lehetséges. Ez azonban nem jelenti azt, hogy ez a modell ne lenne hasznos: alapjául szolgál a rendszerek közötti kommunikáció absztrakt strukturálásának és szisztematikus implementációjának.

Egy másik létező referenciamodell a CASE-környezetek (ECMA, 1991; Brown *és mások*, 1992) referenciamodellje, amely öt olyan szolgáltatáscsoportot azonosít, amelyet egy CASE-környezetnek biztosítania kell. Szintén biztosítania kell az ezen szolgáltatásokat használó egyedi CASE-eszközök számára „plug-in" lehetőségeket. A CASE-referenciamodell a 11.12. ábrán látható, szolgáltatásainak öt szintje a következő:

1. *Adattároló szolgáltatások.* Eszközöket biztosítanak az adatelemek és a közöttük lévő kapcsolatok tárolására és kezelésére.
2. *Adatintegrációs szolgáltatások.* Csoportok kezelésére és a közöttük lévő kapcsolatok létesítésére szolgáló eszközöket biztosítanak. Az adattároló szolgáltatásokkal együtt az adatintegráció alapját biztosítják a környezetben.
3. *Feladatkezelő szolgáltatások.* A folyamatmodellek definiálására és bevezetésére biztosítanak eszközöket. Támogatják a folyamatintegrációt.
4. *Üzenetszolgáltatások.* Az eszköz-eszköz, eszköz-környezet és környezet-környezet kommunikáció számára biztosítanak lehetőségeket. Támogatják a vezérlés integrációját.
5. *Felhasználói felület szolgáltatások.* A felhasználói felület fejlesztéshez biztosítanak eszközöket. A megjelenítés integrációját támogatják.



11.12. ábra. CASE-környezetek ECMA-referenciaarchitektúrája

Ez a referenciamodell azt mutatja meg, hogy egy adott CASE-környezetbe mi tartozhat bele. Fontos kihangsúlyozni, hogy egy referenciaarchitektúrának nem minden jellemzője jelenik meg a tényleges architekturális tervekben. Ez azt jelenti, hogy egy rendszertervvel kapcsolatban feltehetünk olyan kérdéseket, hogy „hogyan biztosítja a rendszer az adattároló szolgáltatást?" és „biztosít-e lehetőséget feladatkezelésre?".

Ezen túlmenően, ennek a referenciaarchitektúrának a fő értéke az integrált CASE-eszközök és -környezetek osztályozásában és összehasonlításában rejlik. Szintén használható az oktatásban az ilyen környezetek kulcsfontosságú jellemzőinek kiemelésére és általános módon történő tárgyalására.

Kulcsfogalmak

- A szoftverarchitektúra a rendszer strukturálásának alapvető keretrendszere. Az alkalmazott architektúra befolyásolja a rendszer tulajdonságait – például a teljesítményt, a védettséget és a rendelkezésre állást.
- Az architekturális tervezési döntések közé az alkalmazás típusára, a rendszer elosztására, az alkalmazandó architekturális stílusokra, valamint az architektúra dokumentálásának és kiértékelésének módjára vonatkozó döntések tartoznak.
- Az architekturális tervezés során különféle architekturális modellek alakíthatók ki, mint például a szerkezeti, vezérlési és felbontási modellek.
- A rendszer szervezeti modelljei magukban foglalják a tárolási modelleket, a kliens–szerver modelleket és az absztrakt gép modelleket. A tárolási modellek az adatokat egy közös tárolóhelyen keresztül osztják meg. A kliens–szerver modellek általában elosztják az adatokat. Az absztrakt gép modellek rétegzettek, minden réteg az alapjául szolgáló réteg által biztosított lehetőségek használatával valósítható meg.
- A felbontási stílusok az objektumorientált és a funkcióorientált felbontást tartalmazzák. A csővezetékelvű modellek funkcionálisak, míg az objektummodellek olyan lazán összekapcsolt egyedeken alapulnak, amelyek kezelik saját állapotukat és műveleteiket.
- A vezérlési stílusok közé a központosított vezérlés és az eseményalapú vezérlés tartozik. A központosított modellekben a vezérlési döntések a rendszer állapotától függenek, míg az eseménymodellekben külső események vezérlik a rendszert.
- A referenciaarchitektúrák a szakterület-specifikus architektúrák tárgyalására és az architekturális tervek kiértékelésére és összehasonlítására használhatók.

További irodalom

Software Architecture in Practice, 2nd ed. Ez a könyv a szoftverarchitektúrákat olyan gyakorlati szempontból tárgyalja, amely nem támaszt túl nagy igényeket a szemléletmóddal szemben és ésszerű üzleti magyarázatát adja az architektúrák fontosságának. (Bass, L. *és mások*, 2003, Addison-Wesley.)

Design and Use of Software Architectures. Habár ez a könyv elsősorban a termékvonal-architektúrákra koncentrál, de az első néhány fejezete remek bevezetőként szolgál a szoftverarchitektúrák tervezésének általános kérdéseibe. (Bosch, J., 2000, Addison-Wesley.)

Software Architecture: Perspectives on an Emerging Discipline. Ez a köny volt az első, amely a szoftverarchitektúrákról szólt, és jól tárgyalja a különböző architekturális jellegzetességeket. (Shaw, M. és Garlan, D., 1996, Prentice-Hall.)

Feladatok

11.1. Magyarázza el, hogy miért lehet szükség a rendszer architektúrájának megtervezésére a specifikáció leírása előtt.

11.2. Magyarázza el, hogy miért léphetnek fel tervezési ellentmondások egy olyan architektúra tervezésekor, amelynek legfontosabb funkcionális követelményei a rendelkezésre állás és a védettség.

11.3. Foglalja össze egy táblázatban a jelen fejezetben bemutatott strukturális modellek előnyeit és hátrányait.

11.4. Rövid indoklással együtt tegyen javaslatot az alábbi rendszereknek megfelelő strukturális modellekre:

- egy vasútállomás automatizált jegykiadó rendszere;
- egy számítógéppel vezérelt videokonferencia-rendszer, amely a résztvevők számára egyidejűleg képes képi, hang- és számítógépi adatokat biztosítani;
- egy padlótisztító robot, amely a viszonylag akadálymentes területek (például folyosók) tisztítását végzi. A robot legyen képes a falak és egyéb akadályok érzékelésére.

11.5. Tervezzen maga által választott modellen alapuló architektúrát a fenti rendszerek számára. Tegyen ésszerű feltevéseket a rendszerkövetelményekre.

11.6. A valós idejű rendszerek általában eseményalapú vezérlési modellt használnak. Milyen körülmények között javasolná egy hívás-visszatérés alapú modell használatát egy valós idejű rendszer esetén?

11.7. Rövid indoklással együtt tegyen javaslatot az alábbi rendszereknek megfelelő vezérlési modellekre:

- kötegelt feldolgozó rendszer, amely a ledolgozott órák és a fizetési mutatók alapján kinyomtatja a fizetési szelvényt és a bankszámla-információkat;

- különböző szállítók által gyártott szoftvereszközök, amelyeknek együtt kell működniük;
- a távirányító jeleire reagáló televízióvezérlő egység.

11.8. Fejtse ki az adatfolyammodell és az objektummodell előnyeit és hátrányait, az eloszthatósággal bezáróan! Tegyük fel, hogy egy alkalmazásnak mind egygépes, mind pedig osztott változatára szükség van.

11.9. Adott két integrált CASE-eszközkészlet és önt arra kérik, hogy hasonlítsa össze őket. Magyarázza el, hogy hogyan tudja használni ehhez az összehasonlításhoz a CASE-referenciamodellt (Brown *és mások,* 1992).

11.10. Szükség van-e önálló „szoftverarchitektúra-tervezőkre", akik a felhasználóktól függetlenül terveznék meg a szoftver architektúráját, amit azután egy szoftvercég implementálna? Mik lennének egy ilyen szakma létrehozásának nehézségei?