

19 KOMPONENSALAPÚ SZOFTVERFEJLESZTÉS

TÉMAKÖRÖK

A fejezet célja, hogy bemutasson egy szoftverfejlesztési folyamatot, amely az újrafelhasználható, szabványosított komponensek kompozícióján alapszik. A fejezetet átolvasva:

- megismerjük, hogy a komponensalapú szoftvertervezés a komponensmodelleken alapuló szabványos komponensek fejlesztésével és ezek alkalmazásrendszerekbe történő beillesztésével foglalkozik;
- megértjük, hogy mit jelent a komponens és a komponensmodell;
- megismerjük a CBSE-folyamat legfontosabb tevékenységeit, és megértjük, hogy miért is kell a követelményeinknél kompromisszumokat kötnünk a komponensek újrafelhasználhatóságának érdekében;
- megértünk néhány, a komponensek kompozíciójának folyamata során felmerülő nehézséget és problémát.

TARTALOM

- 19.1. Komponensek és komponensmodellek
- 19.2. A CBSE-folyamat
- 19.3. Komponenskompozíció

Amint az előző fejezetben írtam, az újrafelhasználás-alapú szoftvertervezés az üzleti és kereskedelmi rendszerek fejlesztésének meghatározó irányvonala lett. Az újrafelhasznált entitások az apró függvényektől egészen a teljes alkalmazás-sokig terjednek. Azonban a közepes méretű programkomponensek újrafelhasználása egészen az utóbbi időkig nehézkes volt. A közepes méretű komponensek szignifikánsan nagyobbak, mint egy objektum vagy egy eljárás, és több funkcionalitással is rendelkeznek, de kisebbek és sokkal specifikusabbak, mint az alkalmazásrendszerek. Szerencsére a jelentősebb szoftverszállítók támogatják a szabványosított fejlesztést, és így ma már ez azt jelenti, hogy a komponensek egy keretrendszeren belül (mint például a CORBA) képesek együttműködni. Ezzel a komponensalapú szoftvertervezésen keresztül lehetővé vált a szisztematikus újrafelhasználás.

A komponensalapú fejlesztés, vagy más néven a komponensalapú szoftvertervezés az 1990-es évek végén bukkant fel a szoftverrendszerek fejlesztésének újrafelhasználás-alapú megközelítéseként. Ennek oka az a csalódottság, hogy az objektumorientált fejlesztés nem váltotta be a hozzá fűzött reményeket a széles körű újrafelhasználás terén. Az egyedi objektumosztályok túlságosan részletezettek és specifikusak voltak, és az alkalmazáshoz fordítási vagy szerkesztési időben kellett hozzákapcsolni őket. Használatukhoz szükség volt az osztályok részletes ismeretére, és ez általában azt jelentette, hogy a forráskódnak hozzáférhetőnek kellett lennie. Ez megnehezítette az üzleti objektumoknak mint újrafelhasználható komponenseknek a használatát. A korábbi, optimista jóslatokkal ellentétben, az egyedi objektumoknak soha nem alakult ki jelentős piaca.

A komponensalapú szoftverfejlesztés (component-based software engineering, CBSE) az a folyamat, amely meghatározza, implementálja és rendszerekbe integrálja a lazán csatolt független komponenseket. Egy jelentős szoftverfejlesztési irányzattá vált, miután a szoftverrendszerek egyre nagyobbakká és komplexebbekké válnak, és az ügyfelek egyre megbízhatóbb szoftvereket akarnak, egyre gyorsabban. Az egyetlen útja, hogy megbirkózzunk a komplexitással és a szoftverek egyre gyorsabb leszállításával, az, hogy a szoftverkomponenseket újraimplementálás helyett újrafelhasználjuk.

A komponensalapú szoftvertervezés fő összetevői a következők:

1. *Független komponensek*, amelyek tökéletesen specifikálva vannak az interfészek által. Egyértelműen el kell különíteni a komponens interfészét és az implementációját, így anélkül cserélhetjük ki a komponens implementációját egy másikra, hogy a rendszer megváltozna.
2. *Komponensszabványok*, amelyek megkönnyítik a komponensek integrációját. Ezek a szabványok komponensmodellekben jelennek meg, továbbá meghatározzák, bár nagyon kis mértékben, hogy a komponensinterfészeket miként kell specifikálni és hogyan tudnak a komponensek kommunikálni. Néhány modell meghatároz olyan interfészeket, amelyeket minden illeszkedő komponensnek implementálnia kell. Amennyiben a komponens illeszkedik a szabványhoz, akkor a műveletei függetlenek lehetnek a programozási nyelvtől. Ugyanazon rendszerbe különböző nyelven íródott komponensek is integrálhatók.
3. *Köztes réteg*, amely szoftveres támogatást nyújt a komponensek integrációjához. A független, elosztott komponensek együttműködéséhez szükségünk van a köztes rétegre, amely a komponensek közötti kommunikációt kezeli. A köztes rétegek [mint például a CORBA (Pope, 1998) amelyet a 12. fejezetben mutattam be] hatékonyan kezelik az alacsony szintű folyamatokat, így lehetővé teszik, hogy csak az alkalmazással kapcsolatos problémákra összpontosítsunk. Ezenfelül egy komponensmodell implementálásánál a köztes réteg támogatást nyújthat az erőforrás-allokációhoz, a tranzakciókezeléshez, a biztonsághoz és a konkurencia kezeléséhez.
4. *Egy fejlesztési folyamat*, amely a komponensalapú szoftvertervezéshez kötődik. Ha egy olyan fejlesztési folyamathoz próbáljuk meg hozzáadni a komponens-

alapú megközelítést, amely a hagyományos szoftver-előállításához kötődik, akkor azt fogjuk tapasztalni, hogy a folyamatban rejlő feltevések korlátozzák a CBSE lehetőségeit. A CBSE fejlesztési folyamat a 19.2. alfejezetben szerepel.

A komponensalapú fejlesztés egyre inkább a legmeghatározóbb irányzattá válik, annak ellenére is, hogy az újrafelhasználható komponensek nem mindig érhetők el. A CBSE alapjául olyan szilárd tervezési irányelvek szolgálnak, amelyek az érthető és karbantartható szoftverek készítését támogatják. A komponensek függetlenek, így nem zavarják egymás működését. Az implementációs részletek rejtve maradnak, ezért a komponensek implementációja anélkül változtatható, hogy megzavarnánk a rendszert. A komponensek kommunikációja jól meghatározott interfészeken keresztül történik, így ha ezeket fenntartjuk, akkor a komponensek kicserélhetők, akár olyanokra is, amelyek további vagy fejlettebb funkcionalitást nyújtanak. Továbbá a komponens-infrastruktúrák olyan magas szintű platformot biztosítanak, amelyek csökkentik az alkalmazásfejlesztés költségeit.

Jóllehet a CBSE igen gyorsan válik a szoftverfejlesztés egyik meghatározó irányzatává, azért számos probléma létezik még:

1. *Komponens megbízhatósága.* A komponensek fekete doboz progamegységek, és a komponens forráskódja nem feltétlenül érhető el a komponens felhasználók számára. Ilyen esetekben a felhasználó honnan tudja, hogy a komponens megbízható-e? A komponens ezenfelül rendelkezhet olyan nem dokumentált hibakezelő móddal, amely a komponens felhasználó rendszer működését veszélyezteti. A nemfunkcionális működése eltérhet a várttól, és a legsúlyosabb esetben, a fekete doboz komponens akár egy trójai program is lehet, ami a rendszer biztonságát megsértő kódot rejt.
2. *Komponens tanúsítvány.* A tanúsítvány esete nagyon hasonló a megbízhatósághoz. Már javasolták, hogy független minősítők hitelesítsék a komponenseket annak érdekében, hogy a felhasználók számára garantálható legyen a komponensek megbízhatósága. Azonban még nem tisztázott, hogy miként lehet ezt elvégezni. Kinek kell fizetnie a minősítésért, ki lesz a felelős azért, ha egy komponens mégsem a tanúsítványnak megfelelően működik, továbbá a minősítők hogyan korlátozhatják saját felelősségüket? Nézeteim szerint az egyetlen járható megoldás az, ha a komponensek esetében garantáljuk, hogy egy formális specifikációnak megfelelnek. Azonban nem úgy tűnik, hogy az ipar hajlandó lenne erre áldozni.
3. *Eredendő tulajdonságok előrejelzése.* Amint azt a 2. fejezetben már kifejtettem, minden rendszer rendelkezik eredendő tulajdonságokkal, és ezen eredendő tulajdonságoknak az előrejelzése és ellenőrzése fontos a rendszer fejlesztési folyamatában. Miután a komponensek nem átláthatók, az eredendő tulajdonságaiknak az előrejelzés különösen nehéz. Következésképpen azt tapasztalhatjuk, hogy miután a komponenseket integráltuk, a létrejött rendszernek lehetnek nemkívánatos tulajdonságai, amelyek korlátozzák a használatát.
4. *Követelmények átformálása.* Általában a rendszer specifikációjában és a tervezési folyamatban engedményeket kell tennünk az ideális követelmények és az

elérhető komponensek között. Jelenleg ezeknek a döntéseknek a meghozatala intuitív folyamat. Ennél azért sokkal strukturáltabb és szisztematikusabb, a helyettesítést elemző módszerekre van szükség, hogy a tervezőknek segíteni tudjunk a komponensek kiválasztásában és konfigurálásában.

A CBSE legfontosabb alkalmazási területe mostanáig a vállalati információs rendszerek, mint például az e-kereskedelmi rendszerek létrehozása volt. Az újr felhasznált komponensek vagy belső fejlesztésűek voltak, vagy ismert, megbízható szállítóktól származtak. Bár néhány kereskedő a komponenseket az interneten keresztül is árulja, a cégek többsége még mindig vonakodik megbízni a kívülről érkező, bináris komponensekben. Így valószínűtlen, hogy a CBSE teljes víziója a specializálódott komponensbeszállítókkal meg tudjon valósulni mindaddig, amíg ezek a problémák meg nem oldódnak.

19.1. KOMPONENSEK ÉS KOMPONENSMODELLEK

Általános megállapodás a közösségben, hogy a komponens egy független szoftveregység, amely más komponensekkel összeilleszthető, hogy egy szoftverrendszer jöjjön létre. Ezen túlmenően azonban néhányan további definíciót is megadtak a szoftverkomponensről. Councill és Heineman (Councill és Heineman, 2001) szerint a *komponens*:

egy szoftver-alkotóelem, amely megfelel egy komponensmodellnek, önállóan kihelyezhető, és módosítás nélkül beilleszthető az összeillesztési szabványnak megfelelően.

Ez a definíció elsősorban a szabványokon alapszik – az a szoftveregység, ami megfelel ezeknek a szabványoknak, a komponens. Szyperski (Szyperski, 2002) azonban a komponens definíciójában nem említi a szabványokat, hanem a komponensek kulcsfontosságú jellemzőire összpontosít:

Egy szoftverkomponens az összeillesztés egy egysége, amely csak szerződészerűen meghatározott interfészekkel és explicit kontextusfüggőségekkel rendelkezik. Egy szoftverkomponens önállóan kihelyezhető, és alkalmas harmadik fél által történő összeillesztésre.

Szyperski továbbá azt is kijelenti, hogy a komponens nem rendelkezik kívülről megfigyelhető állapottal. Ez azt jelenti, hogy a komponensek másolatai megkülönböztethetetlenek. Azonban néhány komponensmodell, mint például az Enterprise Java Bean modell, megengedi az állapottal rendelkező komponenseket, így ezek nyilvánvalóan nem felelnek meg Szyperski komponensdefiníciójának. Mivel az állapotmentes komponenseket egyszerűbb használni, úgy gondolom, hogy a CBSE befogadhatja mind az állapottal rendelkező, mind az állapotmentes komponenseket.

Ami viszont a fenti definíciókban közös, hogy a komponensek önállóak és a rendszerek összeállításának alapvető egységei. Az én nézőpontom szerint a komponensek teljes definíciója két ajánlás együtteséből származtatható. A 19.1. ábrán a CBSE-folyamatban használható komponenseknek az általam legfontosabbnak vélt jellemzői láthatók.

Komponens jellemzője	Leírás
Szabványosított	A komponens szabványosítása azt jelenti, hogy a CBSE-folyamatban használt komponensek megfelelnek valamely szabványosított komponensmodellnek. Ez a modell meghatározhat komponensinterfészeket, komponens-metaadatokat, -dokumentációt, -kompozíciót és -kihelyezést.
Önálló	A komponensnek függetlennek kell lennie – lehetővé kell tennie, hogy más specifikus komponens nélkül lehessen összeilleszteni és kihelyezni. Amennyiben a komponensnek szüksége van külsőleg biztosított szolgáltatásokra, akkor azokat explicit módon kell beállítani a „szükséges” interfészek specifikációjában.
Összeilleszthető	Annak érdekében, hogy egy komponens illeszthető legyen, minden külső interakciónak nyilvánosan meghatározott interfészekben keresztül kell történnie. Továbbá az önmagáról szóló információkat (mint például a metódusai és attribútumai) kívülről elérhetővé kell tennie.
Kihelyezhető	A kihelyezhetőség érdekében a komponensnek zártnak kell lennie, valamint képesnek kell lennie arra, hogy önálló entitásként működjön egy olyan komponensplatformon, amely megvalósítja a komponensmodellt. Ez általában azt jelenti, hogy a komponens bináris, és nem szükséges lefordítani a kihelyezés előtt.
Dokumentált	A komponenseknek teljesen dokumentáltaknak kell lenniük, hogy a potenciális felhasználók eldönthessék, hogy a komponens megfelel-e igényeiknek. Az összes komponensinterfészt szintaktikailag, és ideális esetben szemantikailag is, specifikálni kell.

19.1. ábra. Komponensek jellemzői

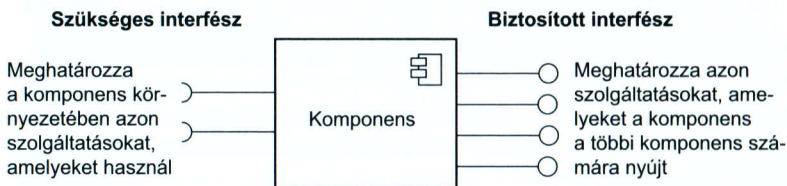
Ezen formális komponensdefiníciók eléggé absztraktak, és nem igazán adnak tiszta képet arról, hogy mit is kell csinálnia egy komponensnek. A legjobb talán úgy gondolni egy komponensre, mint egy önálló szolgáltatóra. Ha egy rendszernek szüksége van valamilyen szolgáltatásra, felkér egy komponenst, hogy biztosítsa ezt a szolgáltatást. Mindezt anélkül teszi, hogy azzal törődne, hogy az adott komponenst milyen programozási nyelven írták meg, illetve hogy az hol fut. Például egy könyvtári rendszer komponense nyújthat olyan kereső szolgáltatást, amellyel a felhasználók különböző könyvtári katalógusokban kereshetnek. Elképzelhetünk olyan komponenst is, amely adatkonverziós szolgáltatást biztosít a különböző grafikus formátumok között (mondjuk például TIFF formátumról JPEG formátumra).

A komponenst szolgáltatóként tekintve az újrafelhasználható komponensek két kritikus jellemzőjét emelhetjük ki:

1. A komponens független, végrehajtható entitás. Forráskódja nem hozzáférhető, így a további rendszerkomponensekkel együtt történő használatbavétele előtt nem kell lefordítani.
2. A komponens által nyújtott szolgáltatások interfészeken keresztül érhetők el, és minden interakció ezeken az interfészeken keresztül történik. A komponens interfésze parametrikus műveletekkel adható meg, belső állapota pedig sohasem látható.

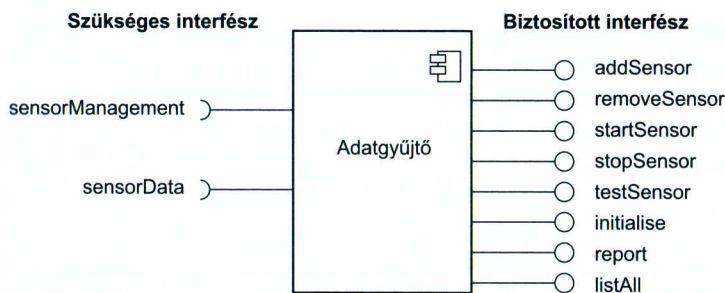
A komponensek interfészeik segítségével adhatók meg, és leggyakrabban úgy képzelhetők el, hogy kétfajta interfésszel rendelkeznek, amint azt a 19.2. ábrán is láthatjuk:

1. *Biztosított* interfésszel, amely a komponens által nyújtott szolgáltatásokat definiálja. A biztosított interfész lényegében a komponens API-ja. Meghatározza a komponens azon metódusait, amelyeket a felhasználó meghívhat. A biztosított interfészt a komponens ikonjából kiinduló és körben végződő vonallal ábrázoljuk.
2. *Szükséges* interfésszel, amely megadja, hogy mely szolgáltatásoknak kell elérhetőnek lenniük a komponens használó rendszerből. Ha ezek nem biztosítottak, akkor a komponens nem működik. Azonban ez nem veszélyezteti a komponens függetlenségét vagy kihelyezhetőségét, mert nem követelmény, hogy a szolgáltatáshoz egy specifikus komponens kellene használni. A szükséges interfészeket a komponens ikonjából kiinduló félkörben végződő vonallal ábrázoljuk. Vegyük észre, hogy a biztosított és a szükséges interfészek úgy illeszkednek egymásba, mint egy gömbcsukló esetén a gömb a csuklóba.



19.2. ábra. Komponensinterfészek

Példaként tekintsük a 19.3. ábrán látható komponens modelljét, amelyet arra terveztek, hogy összegyűjtse és egybevesse a szenzoroktól érkező információkat. Időközönként önállóan végrehajtódik az adatok begyűjtése, és beérkező kérés esetén az összegyűjtött adatokat átadja a hívó komponensnek. A biztosított interfész tartalmazza azokat a metódusokat, amelyekkel szenzorokat lehet felvenni, eltávolítani, elindítani, leállítani és tesztelni. Ezenfelül rendelkezik még jelentéskészítő metódusokkal is (*report* és *listAll*), amelyek a begyűjtött adatokat és a szenzorok konfigurációját szolgáltatják. Bár az ábrán nincs jelölve, metódusok természetesen rendelkeznek paraméterekkel, amelyekkel meghatározhatók a szenzorok helyei és ehhez hasonló dolgok.



19.3. ábra. Egy adatgyűjtő komponens modellje

Az adatgyűjtő komponens megköveteli a szenzoroktól, hogy egy menedzsment- és egy adatinterfészt biztosítsanak. Ezek olyan paraméterekkel rendelkeznek, amelyek meghatározzák a műveletet, illetve a begyűjtendő adatot. A szükséges interfészt pedig szándékosan terveztem úgy, hogy ne tartalmazzon specifikus műveleteket, mint amilyen például a tesztelés. A minél absztraktabb szükséges interfész lehetővé teszi, hogy ebben az esetben az adatgyűjtő komponenset különböző interfészekkel rendelkező komponensekhez is használni tudjuk. A hardver-specifikus szenzorinterfészek és az adatgyűjtő komponens között pedig illesztő komponenseket fogunk interfészként használni.

Az objektumosztályok metódusai egyértelműen hasonlítanak a komponens-interfészekben meghatározott metódusokhoz. Akkor viszont mi a különbség a komponensek és az objektumok között? A komponenseket általában objektumorientáltan fejlesztik, de számos fontos dologban különböznek az objektumoktól:

1. *A komponensek kihelyezhető entitások.* Ez azt jelenti, hogy nem fordítódnak bele az alkalmazói programokba, hanem közvetlenül a futtató platformon kerülnek telepítésre. Az interfészeik által meghatározott metódusok és attribútumok ezután a többi komponens által hozzáférhetővé válnak.
2. *A komponensek nem határoznak meg típusokat.* Egy osztály egy absztrakt adattípust határoz meg, és az objektumok ennek a típusnak a példányai. A komponens egy példánynak tekinthető, és nem egy sablonnak, amit felhasználhatnánk példányok létrehozására.
3. *A komponensimplementációk rejtettek.* A komponensek interfész-specifikációik által pontosan definiáltak, legalábbis elviekben. Az implementáció rejtve marad a felhasználók előtt. A komponenseket általában bináris egységként szálítják le, így a komponens vevőjének nincs hozzáférése az implementációhoz.
4. *A komponensek nyelvfüggetlenek.* Az objektumosztályoknak követniük kell az adott objektumorientált programozási nyelv szabályait, és általában csak az adott nyelven készült osztályokkal tudnak együttműködni. Jóllehet a komponenseket többnyire objektumorientált nyelveken (például Java) implementálják, azonban implementálhatók nem objektumorientált nyelveken is.
5. *A komponensek szabványosítottak.* Az objektumosztályoktól eltérően, amelyeket tetszőlegesen implementálhatunk, a komponenseknek bizonyos komponensmodelleknek kell megfelelniük, ami megszorításokkal jár az implementációjukra nézve.

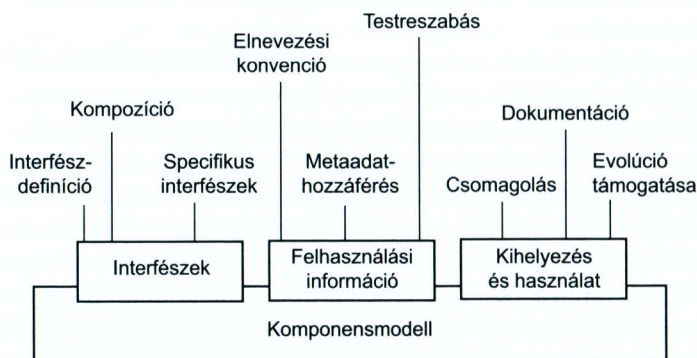
19.1.1. Komponensmodellek

Egy komponensmodell a komponens implementációs, dokumentációs és kihelyezési szabványainak definíciója. A szabványok a komponens fejlesztőinek szólnak, hogy biztosítani lehessen a komponensek együttműködését. Ezenfelül természetesen a komponenseket futtató infrastruktúrát üzemeltetőknek is szólnak, akik olyan köztes szoftvereket biztosítanak, amelyek lehetővé teszik a komponensek működését. Számos javaslat készült komponensmodellekre, de ezek közül a legjelentősebb modell az OMG CORBA komponensmodellje, a Sun Enterprise Java Bean modellje és a Microsoft COM+ modellje (Blevins, 2001; Ewald, 2001; Wang és mások, 2001).

A CBSE-folyamatban használt specifikus infrastruktúra-technológiák, mint például a COM+ és az EJB, nagyon összetettek. Következésképpen nagyon nehéz ezeknek a technológiáknak a leírása anélkül, hogy bele ne mennénk a mögöttük meghúzódó irányzatok alapfeltevéseibe és az alkalmazott interfészek implementációs részleteibe. Azonban itt most az aprólékos részletek helyett igyekszem a komponensmodellek alapvető elemeire összpontosítani.

Egy ideális komponensmodell alapvető elemeit részletesen Weinreich és Sametinger (Weinreich és Sametinger, 2001) tárgyalja. A modellelemek rövid összefoglalását a 19.4. ábrán láthatjuk. A diagramon megfigyelhetjük, hogy a komponensmodell elemei csoportosíthatók, vannak a komponensinterfészre vonatkozó elemek, a komponensek programokban történő használatára vonatkozó információt tartalmazó elemek és a komponens kihelyezésére vonatkozó elemek.

A komponens meghatározó elemek a komponens interfészei. A komponensmodell meghatározza az interfészek specifikációjának módját és azokat az elemeket (például a műveletek nevei, paraméterei és kivételei), amelyeket az interfész definíciójában meg kell adni. A modellnek ezenfelül meg kell határoznia az interfész definíciójához használható nyelvet (az interfész-definíciós nyelvet, IDL). A CORBA és a COM+ esetében ez egy specifikus IDL, az EJB esetében annak Java-specifikussága miatt a Javát alkalmazták mint IDL-t. Néhány komponensmodell



19.4. ábra. Egy komponensmodell alapvető elemei

olyan specifikus interfészeket követel meg, amelyeket a komponenseknek definiálniuk kell. Ezek az interfészek használhatók a komponens-infrastruktúra szabványosított szolgáltatásaival történő kompozícióra, mint például a biztonság és a tranzakciókezelés.

A komponensek távoli elérhetősége és eloszthatósága érdekében egyedi nevet vagy leírót kell hozzájuk rendelni. A COM+ esetén ez egy egyedi 128 bites azonosító. A CORBA komponensmodelljében és az EJB modelljében pedig egy internetes tartománynévből induló hierarchikus név. A komponens metaadata magáról a komponensről hordoz információkat, mint például az interfészei és attribútumai. A metaadat azért is fontos, mert a komponens alkalmazói megtalálhatják, hogy milyen szolgáltatásokat nyújt, és milyeneket igényel a komponens. Ezen komponens-metaadatokhoz történő hozzáférésre alapesetben a komponensmodell-implementációk specifikus lehetőségeket biztosítanak (mint például az ön-elemző interfész a Java nyelvben).

A komponensek általános entitások, így kihelyezésük során a sajátos alkalmazási környezetüknek megfelelően testre kell szabni őket. Például a 19.3. ábrán már bemutatott adatgyűjtő szenzort beállíthatjuk úgy, hogy egy szenzorsor esetében maximum tíz szenzort tartalmazhasson. A komponensmodelleknek ezért meg kell határozniuk, hogy a bináris komponensek miként konfigurálhatók az egyes kihelyezési környezetekben.

A komponensmodellek nagyon fontos részét képezi annak a definiálása, hogy a komponenseket a kihelyezéshez hogyan kell csomagolni mint önálló, futtatható entitásokat. A komponensek önállóságából következik, hogy mindazon dolgokkal együtt kell összecsomagolni őket, amelyeket a komponens-infrastruktúra nem szolgáltat, vagy amelyek a szükséges interfészek által nincsenek definiálva. A kihelyezési információ tartalmazza a csomag összetételét és annak bináris felépítését.

Az új követelmények megjelenésével azonban elkerülhetetlen, hogy a komponenseknek ne kelljen változniuk vagy éppen cserélődniük. A komponensmodelleknek éppen ezért szükséges olyan szabályokat tartalmazniuk, amelyek szabályozzák, hogy a komponensek mikor és hogyan cserélhetők le. Végezetül a komponensmodellnek meg kell határoznia, hogy milyen komponensdokumentáció szükséges. Ennek segítségével lehet a komponenseket megtalálni, és eldönteni, hogy megfelelők-e.

A komponensmodellek azonban nem csak szabványok, egyben a köztes szoftverek alapját is jelentik, amelyek a komponensek futtatásának lehetőségét biztosítják. Weinreich és Sametinger (Weinreich és Sametinger, 2001) az operációs rendszerek analógiáját használják a komponensmodellek magyarázatára, miután az operációs rendszerek általános szolgáltatások egy halmazát biztosítják, amelyeket az alkalmazások felhasználhatnak. A komponensmodellek a komponensek számára összehasonlítható osztott szolgáltatásokat nyújtanak. A 19.5. ábrán néhány olyan szolgáltatást láthatunk, amelyeket például egy komponensmodell implementációja biztosíthat.

Horizontális szolgáltatások		
Komponens-kezelés	Tranzakció-kezelés	Erőforrás-kezelés
Konkurencia	Perzisztencia	Biztonság

Platformszoftárvételek			
Címzés	Interfész-definíció	Kivétel-kezelés	Komponens-kommunikáció

19.5. ábra. Egy komponensmodell szolgáltatásai

A komponensmodellek implementációi által nyújtott szolgáltatások két kategóriába sorolhatók:

1. *Platformszoftárvételek.* Ezek az alapvető szolgáltatások lehetővé teszik a komponensek egymás közötti kommunikációját. A komponensmodell-platformra példa a CORBA. A platformszoftárvételeket a 12. fejezetben tárgyalom.
2. *Horizontális szolgáltatások.* Ezek alkalmazásfüggetlen szolgáltatások, amelyek nagy valószínűséggel a különböző komponensek révén lesznek felhasználva. Ezek a szolgáltatások egyrészt csökkentik a komponensek fejlesztési költségeit, másrészt elkerülhető a komponensek inkompatibilitása.

Annak érdekében, hogy a komponens-infrastruktúra által nyújtott szolgáltatásokat igénybe tudjuk venni, a komponensek egy előre meghatározott, szabványosított tárolóba kerülnek. A *tároló* interfészek halmaza, amelyeket arra használunk, hogy a támogatott szolgáltatások implementációjához hozzáférjünk. A komponensnek a tárolóban történő elhelyezése automatikusan biztosítja a szolgáltatások elérését. A komponensinterfészek önmagukban sosem közvetlenül elérhetők a többi komponens által, a hozzáférés a tárolón keresztül történik.

19.1.2. Újrafelhasználható komponensek fejlesztése

A CBSE-folyamat hosszú távú víziójában olyan komponensellátók jelennek meg, akiknek az üzlet az újrafelhasználható komponensek eladásában és fejlesztésében rejlik. Amint már említettem, a bizalom problémája azt jelenti, hogy a komponensek nyílt piaca még nem alakult ki, és az újrafelhasznált komponensek többségét cégen belül fejlesztették ki. Az újrafelhasználható komponenseket nem specifikusan fejlesztették, hanem olyan, már meglévő komponenseken alapulnak, amelyeket már implementáltak más alkalmazásrendszerekben.

Általában véve a belső fejlesztésű komponensek nem azonnal újrafelhasználhatók. Olyan alkalmazásspecifikus jellemzőket és interfészeket tartalmazhatnak, amelyek más alkalmazásokban valószínűleg nem lennének szükségesek. Éppen ezért ezeket a komponenseket át kell vennünk, és ki kell terjesztenünk, hogy általánosabb, és így még inkább újrafelhasználható verziót készíthessünk. Ennek

nyilvánvalóan megvannak a maga költségei. Ezért először el kell döntenünk, hogy egy komponens újrafelhasználható-e, másrészt az újrafelhasználással megtakarított költségek kiegyenlítik-e a komponens újrafelhasználhatóvá alakításának költségeit.

Az első kérdés megválaszolásához el kell tudnunk dönteni, hogy a stabil szakterületi absztrakciók közül a komponens implementál-e egyet vagy esetleg akár többet is. A stabil szakterületi absztrakciók az alkalmazás szakterületének alapvető koncepciói, amelyek lassan változnak. Például egy banki rendszer esetében a szakterület absztrakciója tartalmazhatja a számlákat, a számlatulajdonosokat és a számlakivonatokat. Egy kórházi menedzsmentrendszer esetében a szakterületi absztrakció kiterjedhet a páciensekre, a kezelési módokra és a nővérekre. A szakterületi absztrakciókat néha üzleti objektumoknak is nevezik. Ha a komponens gyakran alkalmazott üzleti objektum vagy kapcsolódó objektumok egy csoportjának az implementációja, akkor jó eséllyel újrafelhasználható.

A költséghatékonyságra vonatkozó kérdésre akkor tudunk válaszolni, ha meg tudjuk állapítani, hogy mekkora a komponens újrafelhasználhatóvá alakításának költsége. Ezek közé a költségek közé tartozik a komponens dokumentálásának, a komponens validálásának és a komponens még általánosabbá alakításának költsége. Egy komponensnek az újrafelhasználhatóbbá tétele során a következő változtatásokat kell elvégezni:

- az alkalmazásspecifikus metódusok eltávolítása;
- a nevek általánosabbra történő átnevezése;
- a funkcionalitás még teljesebbé tétele érdekében új metódusok felvétele;
- a metódusok kivételkezelésének következetessé tétele;
- egy „beállító” interfész felvétele, hogy a komponens adaptálható legyen a különböző szituációkban;
- a függetlenség növelésének érdekében a szükséges komponensek integrálása.

A kivételkezelés problémája különösen nehéz. Elvben minden kivételnek a komponens interfészének részét kellene képeznie. A komponensek pedig önmaguk kivételeit nem kezelhetik, mert minden alkalmazás rendelkezik a saját kivételkezelési elvárásaival. A komponensek esetében ezért helyesebb, ha meghatározzák a várható kivételeket, és ezeket az interfész részeként teszik elérhetővé. Például egy verem-adatstruktúrát megvalósító egyszerű komponens esetében a komponensnek detektálnia kell és nyilvánosságra kell hoznia a verem túlcsoordulásakor, illetve alulcsordulásakor fellépő kivételeket. A gyakorlatban azonban a komponensek biztosíthatnak néhány lokális kivételkezelést, amelynek a megváltoztatása súlyos következményekkel járhat a komponens funkcionálisára.

Mili és mások (Mili és mások, 2002) a költségek becslésének számos módját tárgyalják, ami a komponensek újrafelhasználhatóvá alakításával jár, továbbá vizsgálják ezeknek a befektetéseknek a várható megterülését is. A komponensek ismételt kifejlesztése helyett azok újrafelhasználásával járó előnyök azonban nem

csupán a termelékenység területén eredményeznek hasznot. A minőség területén is előnyökkel járnak, mert egy újrafelhasznált komponens sokkal megbízhatóbb lehet, illetve a piacra kerülése is előnyösebb lehet. Ezek azok a megnövekedett visszatérülések, amelyek a szoftver gyorsabb kihelyezéséből származnak. Mili és mások különböző formulákat mutatnak ezen visszatérülések becslésére, amint azt a 26. fejezetben bemutatott COCOMO-modell is teszi (Boehm és mások, 2000). Azonban a formulák paramétereit nehéz pontosan megbecsülni, továbbá a formulákat a helyi körülményeknek is megfelelően kell adaptálni. Ezen faktorok miatt sejtésem szerint csak igen kevés szoftverprojekt-vezető számára jelenti annak a lehetőségét, hogy a formulákban megbízzon.

Nyilvánvalóan a komponens újrafelhasználhatósága függ az alkalmazási szakterülettől és a funkcionalitástól is. Amikor egy komponenst általános jelleggel ruházunk fel, akkor növeljük az újrafelhasználhatóságát is. Normál esetben ez azt jelenti, hogy a komponens több művelettel fog rendelkezni, és a komplexitása is nőni fog, ami a komponenst nehezebben érthetővé és használhatóvá teszi.

A komponensek alkalmazhatósága és újrafelhasználhatósága között szükség-szerű eltérések vannak. A komponens újrafelhasználhatóvá tétele magában foglalja az olyan műveletekkel rendelkező általános interfészek halmazát, amelyek gondoskodnak a komponens minden körülmény közötti használhatóságáról. A komponens alkalmazhatósága pedig azt jelenti, hogy olyan egyszerű, minimális interfészeket biztosítunk, amelyeket könnyű megérteni. Az újrafelhasználhatóság növeli a komplexitást, és ezért csökkenti a komponens érthetőségét. Ezért sokkal nehezebb lehet eldönteni, hogy a komponenst mikor és hogyan kell alkalmazni. Amikor egy komponenst újrafelhasználhatóként tervezünk, akkor kompromisszumokat kell kötnünk az általánosság és az érthetőség között.

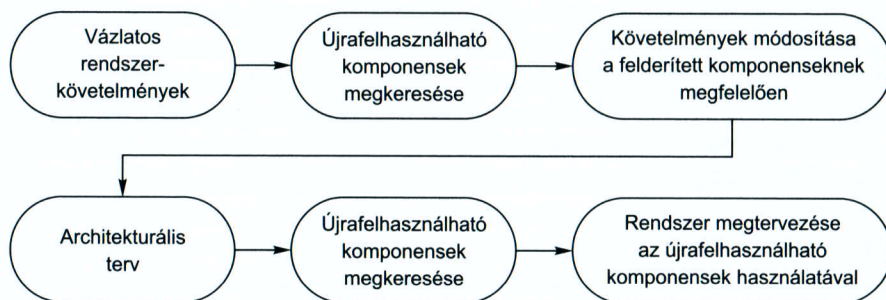
A komponensek egy másik fontos forrását alkotják a meglévő ősrendszerek. Amint arról már a 2. fejezetben szó volt, ezek azok a rendszerek, amelyek fontos üzleti funkciókat hajtanak végre, de elavult szoftvertechnológiával készültek. Emiatt új rendszerekben nehéz lehet az alkalmazásuk. Azonban ha ezeket a régi rendszereket átalakítjuk komponensekké, akkor a funkcionalitásukat újra felhasználhatjuk az új alkalmazásokban.

Természetesen az ősrendszereknek nincsenek egyértelműen definiált szükséges és biztosított interfészei. Ezért a komponensek újrafelhasználhatóvá tételéhez be kell vezetnünk egy csomagolót, amely meghatározza a komponens interfészeit. A csomagoló elrejtja a mögötte rejlő kód komplexitását, és interfészt biztosít a külső komponensek számára a szolgáltatásokhoz történő hozzáféréshez. Természetesen ez a csomagoló igen komplex szoftverrész, miután az ősrendszer funkcionalitásához kell hozzáférnie. Azonban egy ilyen csomagolónak a kifejlesztési költsége még mindig sokkal olcsóbb, mint magának az ősrendszernek az újrainplementálási költsége.

19.2. A CBSE-FOLYAMAT

Amint azt már a bevezetőben is sugalltam, a komponensek sikeres újrafelhasználásához szükséges a fejlesztési folyamatot összekapcsolni a CBSE-folyamattal. Az ilyen jellegű folyamatok struktúráját a 4. fejezetben tárgyaltam. A 19.6. ábrán a CBSE-folyamaton belüli legfontosabb résztvevőket láthatjuk, amelyek között léteznek olyanok, amelyek végrehajtása hasonló módon történik, mint más szoftverfolyamatokban, ilyen például a felhasználói elvárások kezdeti feltérképezése. Azonban az eredeti szoftverfejlesztésen alapuló szoftverfolyamatok és a CBSE-folyamat között az alábbi lényeges különbségek találhatók:

1. A felhasználói követelmények kezdetben nem részletesen, hanem csak körvonalazva kerülnek meghatározásra, valamint a résztvevőket arra kell biztatni, hogy amennyire csak lehetséges, legyenek rugalmasak a követelményeik meghatározásánál. Ennek az az oka, hogy a túl specifikus követelmények miatt jelentősen csökkenhet azon komponensek száma, amelyek megfelelhetnek ezeknek a követelményeknek. Azonban az inkrementális fejlesztéstől eltérően, a követelmények teljes listájára szükségünk van, mert csak így tudjuk megtalálni a lehető legtöbb komponenst az újrafelhasználáshoz.
2. A követelmények már a folyamat elején finomulni és módosulni fognak az elérhető komponensek függvényében. Amennyiben a felhasználói követelményeket nem tudjuk kielégíteni a rendelkezésre álló komponensekkel, akkor át kell beszélünk azon kapcsolódó követelményeket, amelyek fenntarthatók. A felhasználók hajlandók lehetnek megváltoztatni az álláspontjukat, ha ez olcsóbb vagy gyorsabb rendszerátadást jelent.
3. A rendszer-architektúra megtervezését követően van még egy további komponenskeresési és tervfinomító tevékenység. Néhány, látszólag használható komponens a többi választott komponens függvényében esetleg nem működik megfelelően, vagy egyszerűen használhatatlanná válik. Jóllehet, hogy a 19.6. ábrán nem látható, ez maga után vonja, hogy további követelményváltoztatások válhatnak szükségessé.
4. A fejlesztés kompozíciós folyamat, amelyben a feltérképezett komponensek integrálásra kerülnek. Ez magában foglalja a komponensek integrálását a



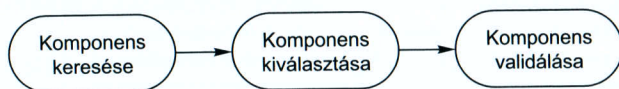
19.6. ábra. A CBSE-folyamat

komponensmodell-infrastruktúrába, és nemegyszer olyan „ragasztó kódnak” a fejlesztését, amely az inkompatibilis komponensek interfészeit összehangba hozza. Természetesen a felhasználható komponensek által nyújtott szolgáltatásokon felül további szolgáltatások is szükségeltethetnek, amelyeket magától értetődően komponensekként célszerű kifejleszteni, hogy a későbbi rendszerekben újrafelhasználhatók legyenek.

Az architektúrális tervezés kimondottan fontos lépés. Az architektúrális tervezés során véglegesen eldönthetjük az alkalmazandó komponensmodellt, bár a rendszerek többségénél ez a döntés már a komponensek keresése előtt megszületik. Amint azt a 11. fejezettől a 13. fejezetig láthattuk, a rendszer magas szintű szerkezetét is kialakítjuk, illetve döntéseket hozunk a rendszer elosztásáról és vezérléséről. Jacobsen és mások (Jacobsen és mások, 1997) úgy találták, hogy egy robusztus architektúra létrehozása kritikus a sikeres újrafelhasználáshoz.

A CBSE-folyamatban az egyetlen egyedi tevékenység a komponens azonosítása. Ez számos résztevékenységet foglal magában, amelyeket a 19.7. ábrán láthatunk. A CBSE-folyamatban két lépés van, ahol a rendszerben esetlegesen felhasználható komponenseket azonosítani kell. A kezdeti lépésben a keresésre és kiválasztásra kell összpontosítani. Meg kell győznünk magunkat, hogy léteznek olyan elérhető komponensek, amelyek megfelelnek az elvárásainknak. Nyilvánvalóan szükséges némi kezdeti ellenőrzés, hogy a komponens megfelelő-e, de részletes tesztelésre nem feltétlenül lesz szükségünk. A későbbi lépésben, miután a rendszer-architektúra megtervezésre került, már több időt kell szentelnünk a komponens validációjára. Biztosnak kell lennünk, hogy az azonosított komponensek valóban megfelelnek az alkalmazásunk számára. Amennyiben mégsem megfelelők, akkor meg kell ismételnünk a keresési és kiválasztási folyamatot.

A komponensek azonosításának első lépése, hogy a helyileg elérhető vagy megbízható szállítóktól származó komponensek után nézünk. A CBSE támogatóinak, mint például Szyperskinek (Szyperski, 2002) a vízióiban olyan életképes komponensvásárterek léteznek, ahol a külső szállítók versenyeznek a komponensek biztosításában. A könyv írásának időpontjában azonban ez még nem valósult meg szignifikáns méretben. A legfőbb oka ennek az, hogy a külső komponensek felhasználóinak olyan kockázatokkal kell szembenézniük, mint például a komponensek nem az elvárt módon történő működése. Ilyen esetekben az újrafelhasználás költsége meghaladja az előnyökét, és így igazán kevés projektvezető hiszi azt, hogy ez a kockázat megéri a befektetést. A másik fő oka, hogy nem alakultak ki ilyen komponens vásárterek, hogy számos komponens speciális alkalmazási szakterületre készült. Ezen szakterületeken pedig nincs megfelelően nagy piac a külső komponensszállítóknak az életképes, hosszú távú üzleti megoldások kialakítására.



19.7. ábra. A komponensek azonosításának folyamata

Következésképpen a komponensek keresése gyakran egy szoftverfejlesztő szervezetre korlátozódik. A szoftverfejlesztő cégek így anélkül építhetik fel saját újrafelhasználható komponens-adatbázisukat, hogy számolniuk kellene a külső szállítók komponenseinek használatával járó kockázatokkal.

Miután a komponenskeresési folyamat azonosította a szóba jöhető komponenseket, a listáról már csak ki kell választani a specifikus komponenseket. Néhány esetben ez egy közvetlen feladat. A listán szereplő komponensek közvetlenül megfelelnek a felhasználói követelményeknek, és így nem léteznek konkurens komponensek, amelyek megfelelnek a követelményeknek. Más esetekben azonban a kiválasztási folyamat sokkal összetettebb is lehet. Nem feltétlenül fog létezni egyértelmű megfeleltetés a követelmények és a komponensek között, akár több komponens is szükséges lehet ahhoz, hogy egy specifikus követelménynek vagy követelmények egy csoportjának megfeleljenek. Szerencsétlen módon még az is előfordulhat, hogy a különböző követelményekhez különböző komponenscsoportok szükségesek, és nekünk kell eldönteni, hogy mely komponens-összeállítás nyújtja a követelmények legjobb lefedését.

Miután a rendszerhez kiválogattuk a szóba kerülhető komponenseket, validálnunk kell azokat annak érdekében, hogy ellenőrizhessük, a viselkedésük megfelel-e az elvártaknak. A szükséges validáció mértéke a komponensforrásoktól függ. Amennyiben egy olyan komponenst használunk, amely egy ismert és megbízható forrásból származik, úgy is dönthetünk, hogy a különálló komponensesztesztelés felesleges, és majd csak akkor teszteljük a komponenst, miután a többi komponenshez hozzáintegráltuk. Másrészt, ha egy ismeretlen forrásból származó komponenst használunk, akkor mindig ellenőrizni és tesztelni kell a komponenst, még mielőtt a rendszerbe bevinnénk.

A komponensvalidáció magában foglalja komponenshez tartozó tesztesetek kifejlesztését (vagy előfordulhat, hogy a komponenssel együtt kaptunk teszteseteket, amelyeket kibővíthetünk), illetve a komponenseszteszt futtatásához egy tesztkörnyezet kifejlesztését. A komponensvalidáció legnagyobb problémája, hogy a komponensspecifikációk nem feltétlenül elég részletesek ahhoz, hogy biztosítsák a komponenseszteszt teljes rendszerének kifejlesztését. A komponensek általában csak informálisan specifikáltak, az egyetlen formális dokumentációjuk csupán az interfészeik specifikációja. Ez viszont nem feltétlenül tartalmaz elegendő információt a tesztek teljes rendszerének kifejlesztéséhez, aminek a segítségével meg tudnánk győződni arról, hogy a komponens által nyújtott interfész az, amire nekünk szükségünk van.

Ennél a lépésnél egy további validációs problémával is szembesülhetünk, nevezetesen a komponens rendelkezhet olyan sajátosságokkal, amelyek zavarók lehetnek a komponens saját igényeinknek megfelelő felhasználásában. Az újrafelhasználható komponensek általában több funkcionalitással rendelkeznek, mint amire szükségünk van. A nem szükséges funkcionalitásokat egyszerűen ignorálhatjuk, de előfordulhat, hogy ezek zavarhatják a többi komponenst vagy magát a teljes rendszert. Néhány esetben a nem szükséges funkcionalitások akár súlyos rendszerhibákhoz is vezethetnek. A 19.8. ábrán olyan szituációt láthatunk, amikor egy újrafelhasznált rendszerben a nem szükséges funkcionalitások katasztrofális hibához vezettek.

Az Ariane 5 hordozórakéta kudarca

Az Ariane 5 hordozórakéta fejlesztésekor a tervezők úgy döntöttek, hogy az Ariane 4 esetében tökéletesen teljesítő tehetetlenségi referenciaszoftvert újrafelhasználják. A tehetetlenségi referenciaszoftver tartja fenn a rakéta stabilitását. A tervezők úgy döntöttek, hogy minden változtatás nélkül újra felhasználják (úgy, ahogyan a komponensek esetében cselekszünk), jöllehet az Ariane 5 számára további kiegészítő funkcionalításra volt szükség.

Az Ariane 5 első fellövésénél a tehetetlenségi navigációs szoftver hibázott 37 másodperc elteltével, és a rakéta irányíthatatlanná vált. A földi irányítás utasította a hordozórakétát az önmegsemmisítésre, és a rakéta rakománya megsemmisült. A későbbi vizsgálatok megállapították, hogy a hibát egy fix pontos szám egészszé történő átalakítása során fellépő numerikus túlszordulási kivétel kezelésének hiánya okozta. Ez a futtatórendszerben a tehetetlenségi referenciarendszer leállítását eredményezte, és ezután már nem lehetett fenntartani a hordozó stabilitását. Az Ariane 4 esetében ilyen hiba sohasem fordulhatott volna elő, mert sokkal gyengébb rendszere volt, és az átalakítandó érték sosem lehetett volna olyan nagy, hogy az átalakítás túlszordulást okozzon.

A hiba olyan kódban lépett fel, amely az Ariane 5 számára nem volt szükséges. Az újrafelhasznált szoftver validációs tesztjeit pedig az Ariane 5 követelményeinek megfelelően állították be. Miután a hibát okozó funkcióval szemben nem voltak követelmények, így tesztek sem készültek hozzá. Következésképpen a szimulációs tesztek során a szoftver hibája sohasem került felszínre.

19.8. ábra. Egy komponensvalidációs hiba

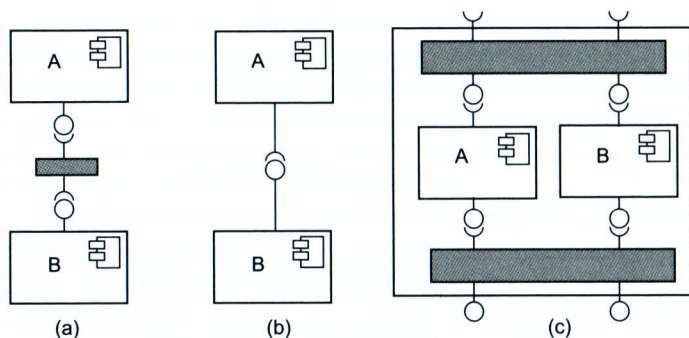
Az Ariane 5 rakétaindító problémáját az okozta, hogy az Ariane 4 szoftverénél megfogalmazott feltevések érvénytelenek voltak az Ariane 5 számára. Ez általános probléma az újrafelhasználható komponenseknél. Eredetileg adott alkalmazási környezethez implementálták, és természetesen az erre a környezetre vonatkozó feltevéseket ágyazták be. Ezeket a feltevéseket csak igen ritkán dokumentálják, így a komponens újrafelhasználásakor lehetetlen olyan teszteket származtatni, amelyek ellenőrizhetnék, hogy a feltevések érvényesek-e még.

19.3. KOMPONENSKOMPOZÍCIÓ

A komponenskompozíció a komponensek összeillesztésének folyamata egy rendszer létrehozása érdekében. Olyan helyzetet feltételezve, amikor rendelkezésre állnak az újrafelhasználható komponensek, a legtöbb rendszer létrehozható lenne az újrafelhasználható komponenseknek egymással, a speciálisan elkészített komponensekkel és a modellkeretrendszer által nyújtott komponens-támogató infrastruktúrával történő kompozícióval. Amint azt a 19.1. alfejezetben láthattuk, az infrastruktúra támogatást biztosít a komponensek kommunikációjához és horizontális szolgáltatásokat is nyújt, mint például felhasználói interfész-szolgáltatások, tranzakciókezelés, konkurencia és biztonság. Az egyes komponensmodellekben a komponensek infrastruktúrával történő integrálásának a lehetőségei dokumentálva vannak, és ebben a szakaszban ezt nem tárgyalom.

A kompozíció nem egy egyszerű művelet, számos típusa létezik (19.9. ábra):

1. *Szekvenciális kompozíció.* Akkor fordul elő, amikor az összetett komponensben az alkotó komponensek végrehajtása szekvenciálisan történik. Ez a 19.9. ábra (a) szituációjának felel meg, ahol az egyes komponensek biztosított interfészei kerültek összefogásra. A komponensek közötti kapcsolat megteremtéséhez plusz kódra van szükség.
2. *Hierarchikus kompozíció.* Akkor fordul elő, amikor egy komponens közvetlenül meghívja a másik komponens által nyújtott szolgáltatásokat. Ez egy olyan szituációnak felel meg, amikor az egyik komponens biztosított interfésze egy másik komponens szükséges interfészével kerül összekapcsolásra. Ez látható a 19.9. ábra (b) részén.
3. *Additív kompozíció.* Akkor fordul elő, amikor két vagy több komponens egy új komponens létrehozásának érdekében kerül összeépítésre. A kompozit komponens interfészei az alkotó komponensek interfészeinek összefogásával állnak elő, szükség esetén a duplikált interfészek eltávolításával. Ez a 19.9. ábra (c) szituációjának felel meg.



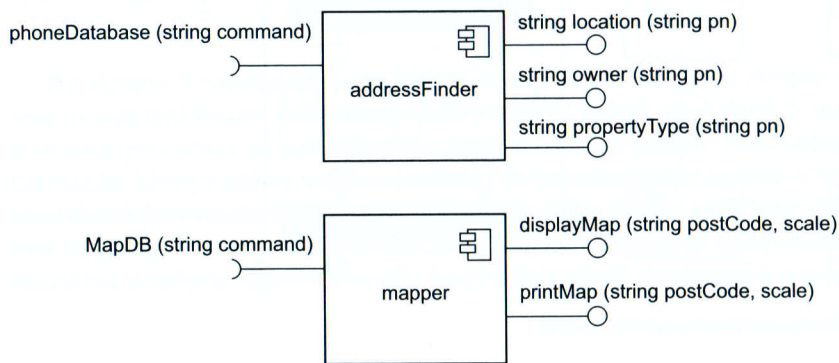
19.9. ábra. Komponenskompozíciók típusai

Egy rendszer létrehozásánál mindhárom komponenskompozíció felhasználható. Viszont minden esetben szükség lehet a komponenseket összekapcsoló „ragasztó kód” megírására. Például a szekvenciális kompozíció esetén az A komponens kimenete tipikusan a B komponens bemenete lesz. Szükségünk lesz olyan közbeni utasításokra, amelyek meghívják az A komponenst, összegyűjtik az eredményét, és meghívják a B komponenst, paraméterként átadva az eredményeket.

Amikor a komponenseket speciálisan kompozícióhoz készítjük, akkor a komponensek interfészeit is megtervezzük, így kompatibilisek lesznek. Ezért a komponenseket könnyedén összefoghatjuk egyetlen egységbe. Azonban amikor a komponenseket az újrafelhasználáshoz önállóan fejlesztjük ki, gyakran szembesülünk az interfészek közötti inkompatibilitással, azaz az összeillesztendő komponensek interfészei nem azonosak. Az inkompatibilitásnak három típusa fordulhat elő:

1. *Paraméter-inkompatibilitás.* A műveletek nevei az interfész mindkét oldalán megegyeznek, de a paramétereik típusa vagy száma eltérő.
2. *Műveletinkompatibilitás.* A szükséges és a biztosított interfészeknél a műveletek nevei eltérőek.
3. *Műveletek nem teljessége.* Egy komponens biztosított interfésze a másik komponens szükséges interfésznek csak egy részhalmaza, vagy fordítva.

Bármely eset is áll fenn, az inkompatibilitás problémájának leküzdéséhez egy olyan adapterkomponenst kell írni, amely az alkalmazott komponensek interfészeit összeegyezteti. Amennyiben ismerjük a felhasználni kívánt komponensek interfészeit, olyan adapterkomponenst kell írni, amely az egyik interfésztől a másira konvertál. Az adapter pontos formája a kompozíció típusától függ. Néhány esetben, ahogyan a következő példában is, az adapter egyszerűen fogja az első komponens eredményeit, és konvertálja arra a formára, amely a másikon már bemenetként használható. Más esetekben előfordulhat, hogy az adaptert az A komponens hívja meg, és maga az adapter hívja meg a B komponenst. Ez utóbbi szituáció akkor alakul ki, ha az A és B komponens ugyan kompatibilis, de az interfészeik paraméterszáma eltérő.



19.10. ábra. Nemkompatibilis komponensek

Az adapterek illusztrálására vizsgáljuk meg a 19.10. ábrán látható komponenseket. Ez egy olyan rendszer része lehet, amelyet sürgősségi szolgáltatásoknál használhatnak. Amikor a sürgősségi operátor fogadja a hívást, a telefonszám az addressFinder komponens bemenetére kerül a cím meghatározásához. Ezután a mapper komponens használatával egy térképet készítenek, hogy ezt az elindított járműnek átküldve, azt a helyszínre irányíthassák. Természetesen a komponensek az itt látottaknál komplexebb interfészekkel is rendelkezhetnek, de ez az egyszerűsített változat illusztrálja az adapterek koncepcióját.

Az első komponens, az addressFinder keresi meg a telefonszámhoz tartozó címet. Ezenkívül még megadhatja a telefonszámhoz tartozó ingatlan tulajdonosát és az ingatlan típusát is. A mapper komponens egy irányítószámot kap (az Egyesült Államokban az ingatlan helyét azonosító négy számjeggyel kiegészített

postai irányítószámot), és megjeleníti vagy kinyomtatja az adott terület utcaterképét adott méretben.

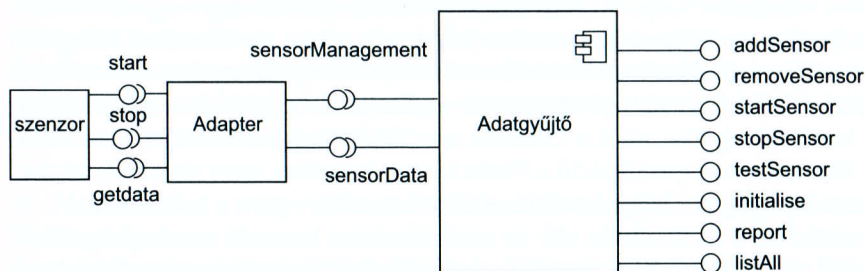
Elvben a komponensek kompozíciója lehetséges, mert az ingatlan helye tartalmazza a posta vagy a kiegészítő kódot. Azonban már nekünk kell elkészíteni azon adapterkomponenst, nevezzük `postCodeStripper`-nek, amely megkapja a helyszín adatait az `addressFinder` komponenstől, és kivágja belőle az irányítószámot. Ezt az irányítószámot használhatjuk ezután a `mapper` bemeneteként, és az utcaszintű térkép már megjeleníthető 1:10 000 méretben. A következő kód illusztrálja, hogy ennek megvalósításához milyen hívási sorozatra van szükségünk:

```
address = addressFinder.location (phonenumber);
postCode = postCodeStripper.getPostCode (address);
mapper.displayMap (postCode, 10000);
```

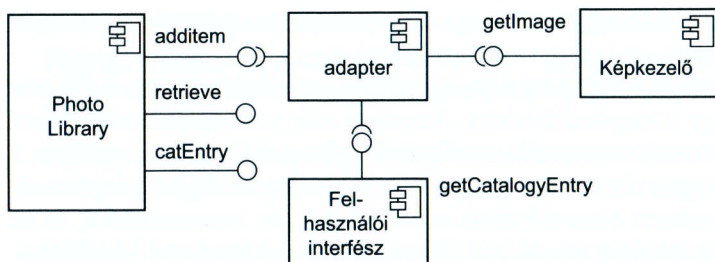
Egy másik eset, amikor adapterkomponenst használhatunk, ha az egyik komponens használni szeretne egy másik komponenst, de a szükséges és a biztosított interfészek között inkompatibilitás áll fenn. Ezt szemlélteti a 19.11. ábra, ahol az adatgyűjtő komponens egy adapter használatával kapcsolódik egy érzékelő komponenshez. Az adapterkomponens összeegyezteti az adatgyűjtő komponens szükséges interfészét a szenzorkomponens biztosított interfészével. Az adatgyűjtő komponens szükséges interfésze egy általános mechanizmusra lett tervezve, ami nem veszi alapul a specifikus szenzorok interfészeit. Előre látható, hogy az adatgyűjtő komponensnek mindig szükség lesz adapterekre ahhoz, hogy a specifikus szenzorokhoz kapcsolódhasson.

A komponenskompozíciónak a megtárgyalása feltételezi azt, hogy a komponensdokumentációk alapján nyilatkozni tudunk a komponensek kompatibilitásáról. Természetesen az interfész-definíciók tartalmazzák a műveletek nevét és paramétereinek típusát, ami alapján bizonyos becsléseket adhatunk a kompatibilitásukról. Azonban a komponens dokumentációjától függ annak eldöntése, hogy az interfészek szemantikailag helyesek-e.

Például tekintsük a 19.12. ábrán látható kompozíciót. Ezeket a komponenseket egy olyan rendszer létrehozására használjuk, amely digitális kameráról tölt le képeket, majd azokat tárolja egy fényképkönyvtárban. A rendszert használók további információkat is megadhatnak a fényképek leírására és katalogizálására.



19.11. ábra. Adapter az adatgyűjtő komponenshez



19.12. ábra. A Photo Library kompozíció

A félreértések elkerülése érdekében nem ábrázolom az összes interfészt, hanem csak egyszerűen azokat, amelyek a komponensek dokumentációs problémájának illusztrálásához szükségesek. A Photo Library interfész metódusai a következők:

```

public void addItem (Identifier pid; Photograph p; CatalogEntry photodesc);
public Photograph retrieve (Identifier pid);
public CatalogEntry catEntry (Identifier pid);
  
```

Tételezzük fel, hogy az addItem metódus dokumentációja a következő a Photo Library-ban:

Ez a metódus a könyvtárhoz ad egy fényképet és a fénykép-azonosítót, illetve a katalógusleíróhoz hozzárendeli a fényképhez.

A leírás részletesnek tűnik, de gondoljunk például a következő kérdésekre:

Mi történik akkor, ha a fénykép-azonosító már hozzá van rendelve egy másik fényképhez a könyvtárban?

A fényképleíró a katalógusbejegyzéshez is ugyanúgy hozzá van rendelve, mint a fényképhez? Azaz, ha törölöm a fényképet, akkor törölöm a katalógusbejegyzést is?

Nincs elegendő információ az addItem informális leírásában ahhoz, hogy ezen kérdésekre válaszolni tudjunk. Természetesen adott a lehetőség, hogy több információt adjunk a metódus természetes nyelvű leírásához, de általában a legjobb megoldás a félreértések elkerülésére, ha valamilyen formális nyelvet használunk az interfész leírására. A 10. fejezetben már említettem, hogy az interfészek leírása azon területek egyike, ahol a formális specifikációk kimondottan hasznosak. A 19.13. ábrán látható specifikáció a Photo Library interfész azon részének leírása, amely az informális leíráshoz további részleteket ad.

A specifikáció a 19.13. ábrán elő- és utófeltételeket használ, amely megadásához az UML (Warmer és Kleppe, 1998) részét alkotó objektummegszorítási nyelven (OCL) alapuló jelöléseket használtam. Az OCL-t az UML objektummodelljé-


```

– A context kulcsszó nevezi meg azt a metódust, amelyre a feltételek vonatkoznak
context addItem

– Az előfeltételek határozzák meg, hogy minek kell igaznak lenni a művelet futtatását megelőzően
pre:   PhotoLibrary.libSize() > 0
        PhotoLibrary.retrieve(pid) = null

– Az utófeltételek határozzák meg, hogy minek kell igaznak lennie a lefuttatás után
post:  libSize () = libSize()@pre + 1
        PhotoLibrary.retrieve(pid) = p
        PhotoLibrary.catEntry(pid) = photodesc

context delete

pre: PhotoLibrary.retrieve(pid) ≠ null;

post: PhotoLibrary.catEntry(pid) = null
        PhotoLibrary.catEntry(pid) = PhotoLibrary.catEntry(pid)@pre
        PhotoLibrary.libSize() = libSize()@pre[em]1

```

19.13. ábra. A Photo Library interfész formális leírása

hez kapcsolódó megszorítások leírására tervezték. Lehetővé teszi olyan állításoknak a kifejezését, amelyeknek mindig igaznak kell lenniük, amelyeknek igaznak kell lenniük, mielőtt a metódus lefutna, és olyanoknak, amelyeknek igaznak kell lenniük azután, hogy a metódus lefutott. Ezeket invariánsoknak, előfeltételeknek és utófeltételeknek nevezzük. A változók művelet előtti értékéhez úgy férhetünk hozzá, hogy a nevük után írjuk a @pre karaktereket. Azaz az

$age = age@pre + 1$

azt jelenti, hogy az *age* változó értékének a művelet végrehajtása után eggyel nagyobbak kell lennie, mint ami a művelet előtt volt.

Az OCL-alapú megközelítések egyre gyakrabban használatosak a szemantikai információknak az UML-modellekhez történő hozzáadására. Az általános megközelítés Meyer Design by Contract (Szerződésalapú tervezés, Meyer, 1992) irányzatából származik, amelyben a kommunikáló objektumok interfészei és kötelezettségei formálisan specifikáltak, és amelyet a futtatórendszer alkalmaz. Meyer ajánlása szerint a Design by Contract használata elengedhetetlen, ha megbízható komponenseket fejlesztünk (Meyer, 2003).

A 19.13. ábra Photo Library-ben található addItem és delete metódusok specifikációját tartalmazza. A specifikálandó metódust a context kulcsszó jelzi, a hozzátartozó elő- és utófeltételeket pedig a pre és post kulcsszavak. Az addItem előfeltételei a következőket fejezik ki:

- Nem létezhet a könyvtárban a feltöltendő fényképpel megegyező azonosítójú fénykép.
- A könyvtárnak léteznie kell – feltételezve azt, hogy egy könyvtár létrehozásával egy elem is kerül bele, így a könyvtár mérete mindig nagyobb, mint egy.

Az addItem utófeltételei a következőket fejezik ki:

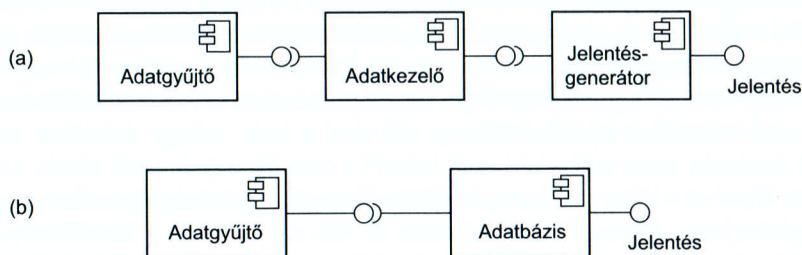
- A könyvtár mérete eggyel nőtt (azaz csak egy hozzáadás történik).
- Ha visszakeressük ugyanazt az azonosítót, akkor az éppen feltöltött fényképet kapjuk vissza.
- Ha a katalógusban keresünk az azonosítót használva, akkor azt a katalógus-bejegyzést kapjuk, amelyet létrehoztunk.

A delete metódus specifikációja további információkat szolgáltat. Az előfeltételei az fejezik ki, hogy egy elem törléséhez az elemnek a könyvtárban kell lennie, és a törlés után a fénykép már nem kereshető vissza, valamint a könyvtár mérete is csökken eggyel. Azonban a törlés nem távolítja el a katalógusbejegyzést – azaz a fénykép törlése után még mindig visszakereshető marad. Ennek oka, hogy a katalógusban szükség lehet olyan információk tárolására, hogy miért kellett a fényképet törölni, vagy hová lett áthelyezve stb.

Amikor egy rendszert komponensek kompozíciójával hozunk létre, úgy tapasztalhatjuk, hogy potenciális ellentmondások létezhetnek a funkcionális és a nemfunkcionális követelmények között annak érdekében, hogy a rendszert, amilyen gyorsan csak lehet, le tudjuk szállítani, illetve hogy olyan rendszert hozzunk létre, amely a követelmények változásával képes lehet a fejlődésre. A következők miatt változhatnak a döntéseink:

- A komponensek mely kompozíciója a leghatékonyabb a rendszer funkcionális követelményeinek megvalósítására?
- A komponensek mely kompozíciója teszi lehetővé a követelmények későbbi változásának az adaptálhatóságát?
- Melyek lesznek az összeállított rendszer eredendő tulajdonságai? Az eredendő tulajdonságok közé fognak tartozni az olyan jellemzők, mint a teljesítmény és a megbízhatóság. Ezeket viszont csak az után fogjuk tudni megállapítani, hogy a teljes rendszert implementáltuk.

Sajnálatos módon számos olyan szituáció létezik, ahol a kompozíciós problémák megoldásai kölcsönösen kizárják egymást. Példának tekintsük a 19.14. ábrán látható helyzetet, ahol a rendszert két alternatív kompozíció segítségével is



19.14. ábra. Adatgyűjtő és jelentéskészítő komponensek

létrehozhatjuk. A rendszer egy adatgyűjtő és jelentéskészítő rendszer, ahol az adatok különböző forrásokból érkeznek, egy adatbázisban kerülnek tárolásra, illetve egy másfajta jelentésben kerülnek összesítésre az előállított adatok.

Az (a) jelű kompozíció előnye, hogy az adatkezelő és jelentéskészítő részek elvannak egymástól választva, azaz a későbbi változtatásokkal szemben nagyobb a flexibilitása. Az adatkezelő rendszer lecserélhető, de ugyanez igaz a jelentéskészítő komponensre is abban az esetben, ha egy olyan jelentést kell készíteni, amit nem tud előállítani. A (b) jelű kompozícióban az adatbázis-kezelő rendszer (például Microsoft Access) beépített jelentéskészítő képességeit használjuk fel. Ennek a kompozíciónak az előnye, hogy kevesebb komponens alkotja, így várhatóan gyorsabb lesz, mert nincs kommunikációs terhelés a komponenseken. Továbbá az adatbázisra vonatkozó adatintegritási szabályok egyúttal a jelentésekre is vonatkoznak, így a jelentések nem tudják helytelenül összeállítani az adatokat. Az (a) jelű kompozíció esetén nem léteznek ilyen megszorítások, így a jelentésekben sokkal nagyobb eséllyel fordulhatnak elő hibák.

Általános értelemben véve elmondható, hogy az a jó és követendő kompozíciós alapelv, amely a vonatkozásokat elválasztja egymástól. Ez az, amire törekedni kell a rendszer megtervezésénél ahhoz, hogy minden egyes komponensnek jól definiált szerepköre legyen, valamint arra az ideális esetre, hogy ezek a szerepkörök ne fedjék át egymást. Továbbá az üzembiztonság és a teljesítmény területén hátrányos lehet, ha összetett komponenseket alkalmazunk.

Kulcsfogalmak

- A komponensalapú szoftvertervezés egy újrafelhasználás-alapú irányzat, amely azt szolgálja, hogy meghatározzunk, implementáljunk és lazán kapcsolódó független komponenseket illesszünk össze rendszerekké.
- A komponens egy szoftveregység, amelynek funkcionalitása és függőségei teljesen meghatározottak a nyilvános interfészei által. A komponensek más komponensekkel illeszthetők össze anélkül, hogy az implementációjukat ez zavarná, illetve kihelyezhetők mint futtatható egység.
- A komponensmodellek a komponensek számára határoznak meg szabványokat, beleértve az interfészsabványokat, felhasználási szabványokat és kihelyezési szabványokat. A komponensmodellek implementációi horizontális szolgáltatásokat biztosítanak, amelyek bármely komponens által felhasználhatók.
- A CBSE-folyamat során össze kell illeszteni a követelménytervezés folyamatát a rendszertervezéssel. A kívánatos követelményekkel szemben engedményeket kell tennünk a létező és elérhető újrafelhasználható komponensek által nyújtott szolgáltatások irányában.
- A komponenskompozíció az a folyamat, amikor a komponenseket egymáshoz „drótozzuk” annak érdekében, hogy egy rendszer jöjjön létre. A kompozíció típusa lehet szekvenciális kompozíció, hierarchikus kompozíció és additív kompozíció.

- Amikor olyan újrafelhasználható komponenseket illesztünk össze, amelyek nem az alkalmazásunkhoz készültek, akkor normális esetben adaptereket vagy „ragasztó kódot” kell írunk a különböző komponensinterfészek összeegyeztetésére.
- Amikor kiválasztjuk a kompozíciót, át kell gondolnunk a rendszertől elvárt funkcionalitást, a nemfunkcionális követelményeket, és meg kell könnyítenünk annak a lehetőségét, hogy a rendszer változtatásával egy komponenst kicserélhessünk egy másikra.

További irodalom

Component-based Software Engineering: Putting the Pieces Together. A CBSE különböző szemléletű cikkeinek gyűjteménye, több szerzőtől. Mint minden gyűjtemény, ez is vegyes, de a komponensekkel történő szoftverfejlesztés általános dolgait jobban lefedi, mint Szyperski könyve. (Heineman, G. T. és Councill, W. T., 2001, Addison-Wesley.)

Component Software: Beyond Object-Oriented Programming, 2nd ed. A CBSE-könyv átdolgozott kiadása átöleli a CBSE-folyamat technikai és nem technikai kérdéseit. A specifikus technológiákat részletesebben tartalmazza, mint Heineman és Councill könyve. Átfogóan tárgyalja a piaci kérdéseket. (Szyperski, C., 2002, Addison-Wesley.)

„Specification, implementation and deployment of components”. Jó bevezetés a CBSE alapjaiba. A CACM ugyanezen kiadása a komponensekről és a komponensalapú fejlesztésről tartalmaz cikkeket. (Crnkovic, I. és mások, *Comm. ACM*, 45(10), October 2002.)

Feladatok

- 19.1. Miért fontos, hogy minden komponens-interakció a szükséges és biztosított interfészek segítségével kerüljön meghatározásra?
- 19.2. A komponensfüggetlenség alapelve azt jelenti, hogy egy komponensnek helyettesíthetőnek kell lennie egy teljesen más úton implementált másik komponenssel. Egy példán keresztül mutassa be, hogy egy ilyen komponenscserének milyen nemkívánatos következményei lehetnek, és ez hogyan vezethet rendszerhibához.
- 19.3. Melyek az alapvető különbségek a webszolgáltatások és a komponensek között (12. fejezet).
- 19.4. Miért fontos, hogy a komponensek komponensmodelleken alapuljanak?
- 19.5. Egy absztrakt adattípust (például egy vermet vagy egy sort) implementáló komponenst felhasználva mutassa meg, hogy miért szükséges az újrafelhasználás érdekében a komponenseket kiterjeszteni és adaptálni.

- 19.6. Magyarázza el, hogy miért nagyon nehéz validálni az olyan újrafelhasználható komponenseket, amelyeknek nem érhető el a forráskódja. Milyen módokon egyszerűsítheti a komponensvalidációt egy formális komponensspecifikáció?
- 19.7. Tervezzon egy újrafelhasználható komponenst a korábbi fejezetekben ismertett LIBSYS-rendszer keresési lehetőségének implementálására, ami nem egy egyszerű kulcsszavas keresője a weboldaloknak. Képesnek kell lennie arra, hogy a felhasználó által megadott különböző könyvtárak katalógusaiban tudjon keresni.
- 19.8. Példák segítségével illusztrálja a szekvenciális, a hierarchikus és az additív kompozíció megvalósításhoz szükséges adapterek különböző típusait.
- 19.9. Tervezze meg azon komponensek interfészeit, amelyek felhasználhatók egy sürgősségi szolgáltatás irányítótermében elhelyezendő rendszernél. Meg kell terveznie a hívásrögzítő komponens interfészeit, amely rögzíti a hívásokat, továbbá a járműkereső komponensét is, amely az irányítószám és a baleset típusa alapján megtalálja a legközelebbi megfelelő járművet, ami a baleset helyszínére irányítható.
- 19.10. Egy független tanúsítványhitelesítő hatóság felállítását javasolják. A szálítókknak el kell küldeniük komponenseiket ehhez a hatósághoz, amely aztán a komponensek megbízhatóságát validálja. Melyek lehetnek egy ilyen hatóságnak az előnyei és a hátrányai?