

32 ASPEKTUSORIENTÁLT SZOFTVERFEJLESZTÉS

TÉMAKÖRÖK

A fejezet célja, hogy bevezetést nyújtson az aspektusorientált szoftverfejlesztésbe, amely a vonatkozások különálló modulokba történő szétválasztásán alapszik. Mire a fejezet végére érünk:

- meg fogjuk érteni, hogy a vonatkozások különválasztása miért jó irányelv a szoftverfejlesztésben;
- tisztában leszünk az aspektusok és az aspektusorientált szoftverfejlesztés alapjául szolgáló fogalmakkal;
- meg fogjuk érteni, hogyan használjunk aspektusorientált megközelítést a követelmények feltárásában, a szoftvertervezésben és a programozásban;
- ismerni fogjuk az aspektusorientált rendszerek tesztelésével kapcsolatos problémákat.

TARTALOM

- 32.1. A vonatkozások szétválasztása
- 32.2. Aspektusok, csatlakozási pontok és vágási pontok
- 32.3. Szoftverfejlesztés aspektusokkal

A nagy rendszerek többségében a programkomponensek és a követelmények közötti összefüggések rendkívül összetettek. Egyetlen követelmény megvalósítása számos komponenst érinthet, illetve egyetlen komponens több követelmény elemeit tartalmazhatja. Ez azt jelenti, hogy a követelmények változásának implementálása sok komponens megértését és módosítását vonhatja maga után. A komponensek újrafelhasználása bonyolult lehet, mivel azok nem csupán egyetlen rendszerabsztrakciót valósítanak meg, hanem más követelményekhez tartozó kódrészleteket is fellelhetünk bennük.

Az aspektusorientált szoftverfejlesztés (aspect-oriented software development, AOSD), amelyet a 18. fejezetben már röviden tárgyaltam, a szoftverfejlesztés egyre inkább előtérbe kerülő megközelítése, amely ennek a problémának a megoldására hivatott, hogy ezáltal a programok újrafelhasználása és karbantartása egyszerűbbé váljon. Az AOSD egy újfajta absztrakcióra, az aspektusra épül. Az aspektusokat más absztrakciókkal (mint a módszerek és az objektumok)

együtt használjuk. Az aspektusok olyan funkciók bezárására szolgálnak, amelyek keresztezik a rendszer más funkcióit, vagy azokkal együtt léteznek. Egy végrehajtható aspektusorientált program az objektumok, módszerek és aspektusok automatikus összeszövésével jön létre. Ez az összeszövés a program forráskódjában található specifikációknak megfelelően történik.

Az aspektusorientált megközelítés legnagyobb előnye, hogy támogatja a vonatkozások szétválasztását. Ahogy azt a 32.1. fejezetben bővebben is kifejttem, a vonatkozások szétválasztása egymástól független elemekre (a különböző vonatkozások egyazon logikai absztrakcióba építése helyett), jó szoftverfejlesztési gyakorlat. Ha a funkcionalitást keresztülszelő vonatkozásokat aspektusokként reprezentáljuk, akkor ezek a vonatkozások egymástól függetlenül megérthetők, módosíthatók és újrafelhasználhatók. Például tegyük fel, hogy a felhasználói azonosítást olyan aspektusként reprezentáljuk, amely egy felhasználónevet és egy jelszót kér. Ez automatikusan beleszőhető a programba bárhol, ahol azonosításra van szükség.

Az aspektusok egyik fontos tulajdonsága, hogy definíciót tartalmaznak arra vonatkozóan, hogy hol kell őket beilleszteni a programba. Ugyanez igaz a funkcionalitást keresztülszelő vonatkozásokat megvalósító kódokra is. Más szavakkal: meghatározhatjuk, hogy a keresztező kódot egy bizonyos módszer meghívása elé, mögé, vagy egy attribútum megváltoztatásának helyére kell-e beszúrni. Lényegében az aspektus beleszőődik a program magjába, és egy nagyobb rendszer jön létre.

Például tegyük fel, hogy adott egy követelmény, mely szerint azonosításra van szükség az adatbázisban található bármilyen személyes adat megváltoztatása előtt. Ehhez egy aspektusban azt kell megadnunk, hogy az aspektust minden olyan módszer meghívása elé be kell szúrni, amely a személyes adatokat frissíti. Később a követelmény megváltozhat úgy, hogy minden egyes adatbázis-frissítés előtt szükség van azonosításra. Ez egyszerűen megvalósítható azáltal, hogy az aspektusban módosítjuk annak definícióját, hogy az azonosítás kódját hol kell beleszőni a rendszerbe. Nem kell végignéznünk a teljes rendszert, hogy megtaláljuk ezen módszerek minden előfordulását. Ennek köszönhetően sokkal kisebb valószínűséggel vétünk hibákat, és kisebb az esélye annak is, hogy véletlenül biztonsági réseket építünk a rendszerbe.

Az aspektusorientáltság kutatásának és fejlesztésének homlokterében az aspektusorientált programozás áll. Olyan aspektusorientált programozási nyelveket fejlesztettek ki, amelyek az aspektusok bevezetésével kiterjesztik az objektumorientált programozást. Ilyen nyelv az AspectJ (Kiczales és mások, 2001; Colyer és Clement, 2005; Laddad, 2003a; 2003b; Colyer és mások, 2005). Az aspektusorientált programozást a vállalatok (például IBM) is kezdi felhasználni a szoftvergyártási folyamataikban (Colyer és Clement, 2005). Azonban mára felismerték, hogy a funkcionalitást keresztülszelő vonatkozások a szoftverfejlesztési folyamat más szakaszaiban is legalább ennyire problematikusak. A kutatók most azt vizsgálják, hogyan lehetne az aspektusorientáltságot a követelmények feltárásában és a szoftvertervezésben alkalmazni, továbbá foglalkoznak az aspektusorientált programok verifikálásának és tesztelésének kérdésével is.

Az aspektusorientált szoftverfejlesztés még nem része a főáramú szoftverfejlesztésnek. Még nem fogalmaztunk meg alapelveket és nem teszteltük az AOSD gyakorlatait, és hogy erre sor kerüljön, még jó néhány év tapasztalatára szükségünk van. Mint minden új technológia esetén, a szószólók inkább az előnyökre koncentrálnak, semmint a hátrányokra és a költségekre. Hiszem, hogy az AOSD nagy hatással lesz a szoftverfejlesztés gyakorlatára, mégis úgy gondolom, hogy sok évnek el kell még telnie addig, hogy problémákat oldanánk meg ezzel a megközelítéssel. Az AOSD-t azért tárgyalom itt, mert a vonatkozások szétválasztása nagyon fontos szemléletmód, és a szoftverrendszerek strukturálásának fontos módszere.

A fejezet hátralévő részében szót ejtek néhány olyan új fogalomról, amelyek az AOSD részei, valamint beszélek az aspektusorientált szemléletmód alkalmazásának előnyeiről és hátrányairól a szoftverfejlesztési folyamat különböző szakaszaiban. Mivel elsődleges célom az AOSD alapjául szolgáló fogalmak tisztázása, nem bocsátkozom részletekbe semmilyen specifikus megközelítéssel vagy aspektusorientált programozási nyelvvel kapcsolatban.

32.1. A VONATKOZÁSOK SZÉTVÁLASZTÁSA

A vonatkozások szétválasztása a szoftvertervezés és implementáció egyik kulcsfontosságú alapelve. Lényegében azt jelenti, hogy úgy kell szerveznünk szoftvereinket, hogy a program minden eleme (osztály, módszer, eljárás stb.) pontosan egy dolgot csináljon. Ha így teszünk, akkor foglalkozhatunk egy adott elemmel anélkül, hogy a többi figyelemmel kísérnénk. A program minden részét megérthetjük, ha ismerjük annak feladatát, nem kell megértenünk más elemeket. Ha változtatásokra van szükség, azok csak kisszámú elemet érintenek.

A vonatkozások szétválasztásának fontosságát már a szoftverfejlesztés történetének korai szakaszában felismerték. A szubrutinokat, amelyek egy kis egységnyi funkcionalitást zártak magukba, az 1950-es években vezették be, és minden rá következő szoftverstrukturálási mechanizmus (mint az eljárások és az osztályok) a vonatkozások szétválasztásának tökéletesítését szolgálták. Azonban ezen mechanizmusoknak problémáik voltak bizonyos fajtájú vonatkozásokkal, mégpedig azokkal, amelyek keresztülmetszenek más vonatkozásokat. Az aspektusokat az ilyen vonatkozások kezelésének egyszerűsítésére vezették be.

Miközben általánosan elfogadott tény, hogy a vonatkozások szétválasztása jó szoftverfejlesztési gyakorlat, annak meghatározása, hogy mit is értünk valójában vonatkozás alatt, igen nehéz. Olykor funkcionális fogalomként definiálják: egy vonatkozás a rendszer funkcionalitásának valamilyen eleme; definiálható azonban sokkal általánosabban is, mint „a program bármely olyan része, amely valamilyen szempontból érdekes vagy lényeges”. A két definíció közül egyik sem igazán alkalmas. A vonatkozások mindenképpen többek egyszerű funkcionális elemeknél, az általános definíciót pedig homályossága teszi gyakorlatilag használhatatlanná.

Véleményem szerint a legtöbb kísérlet a vonatkozások definiálására azért problematikus, mert megpróbálja a vonatkozásokat kapcsolatba hozni a programokkal. Valójában, ahogy azt Jacobsen és Ng (Jacobsen és Ng, 2004) tárgyalják, a vonatkozások a rendszerkövetelményeknek és a megrendelők prioritásainak visszatükröződései a rendszerben. A rendszerhatékonyság lehet egy vonatkozás, mivel a felhasználók gyors választ várnak egy rendszertől; néhány megrendelő abban lehet érdekelt, hogy a rendszer tartalmazzon bizonyos funkciókat; az olyan vállalatok, amelyek támogatnak egy rendszert, fontos vonatkozásnak találhatják a karbantarthatóságot. A vonatkozás tehát valami, ami érdekes vagy fontos egy megrendelőnek vagy megrendelők egy csoportjának.

A vonatkozás fogalmának tisztázása után láthatjuk, miért jó gyakorlat egy olyan megközelítés, amely a vonatkozásokat különböző programelemekben valószínűsíti meg. Ilyen módon egyszerűbb a követelményként vagy követelmények egy csoportjaként megfogalmazott vonatkozások figyelése, valamint az őket megvalósító programkomponensek nyomon követése. Ha a követelmények megváltoznak, egyértelműen meghatározható, hogy a program mely részein kell módosításokat végrehajtani.

A megrendelőket érdeklő vonatkozásoknak több típusa létezik:

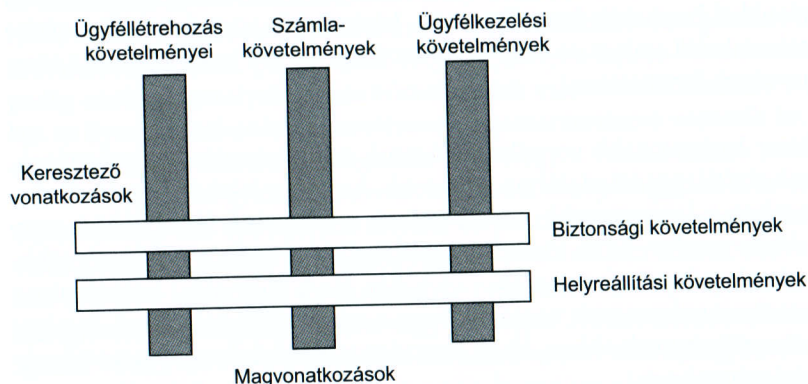
1. A funkcionális vonatkozások azzal kapcsolatosak, hogy valamilyen funkció belekerüljön a rendszerbe. Egy mozdonyokat irányító rendszerben például funkcionális vonatkozás a mozdony lefékezése.
2. A szolgáltatás minőségével kapcsolatos vonatkozások azok, amelyek a rendszer nemfunkcionális viselkedésére vonatkoznak. Ezek magukban foglalják például a hatékonyságot, megbízhatóságot, rendelkezésre állást.
3. A politikai vonatkozások olyan általános intézkedésekkel állnak kapcsolatban, amelyek a rendszer használatát vezérlik. A politikai vonatkozások közé tartoznak a biztonsági vonatkozások, és az üzleti szabályokkal kapcsolatos vonatkozások.
4. Rendszervonatkozások azok, amelyek a rendszer egészének tulajdonságaival hozhatók kapcsolatba. Ilyenek a karbantarthatóság és a konfigurálhatóság.
5. A szervezeti vonatkozások olyan vonatkozások, amelyek a szervezeti célokkal és prioritásokkal hozhatók összefüggésbe. Idetartozik a rendszer elkészítése költségvetésen belül, valamint a meglévő szoftvervagyon használata és a szervezet hírnevének fenntartása.

Egy rendszer legfontosabb vonatkozásai azok a funkcionális vonatkozások, amelyek annak elsődleges feladatához tartoznak. Azaz egy kórház betegnyilvántartó rendszerében a legfontosabb vonatkozások a rekordok létrehozása, szerkesztése és visszanyerése. Ezen vonatkozások mellett a nagy rendszerek többnyire tartalmaznak másodlagos vonatkozásokat is. Ezek megvalósíthatnak olyan funkciókat, amelyek információt osztanak meg a magvonatkozásokkal, és olyanokat, amelyek szükségesek ahhoz, hogy a rendszer megfelelhessen a nemfunkcionális követelményeknek.

Egyszerű példaként képzeljünk el egy olyan rendszert, amellyel szemben támasztott egyik követelmény, hogy konkurens hozzáférést kell biztosítania egy megosztott pufferhez. A 15. fejezetben, amely a valós idejű rendszereket tárgyalja, szót ejtettem egy ilyen rendszerről (15.14. ábra). Ez a megosztott puffer része egy olyan adatfeldolgozó rendszernek, amelyben egy termelő adatokat helyez el a pufferben, egy fogyasztó pedig adatokat vesz ki onnan. A legfontosabb vonatkozás itt a megosztott puffer fenntartása, azaz a magfunktionalitás az adatok hozzáadása a pufferhez, illetve az adatok kivétele. Azonban, annak érdekében, hogy a termelő és a fogyasztó folyamatok ne kerüljenek konfliktusba egymással, megjelenik egy másodlagos vonatkozás, a szinkronizáció. A rendszert úgy kell megtervezni, hogy a termelő ne ír hasson felül olyan adatot, amelyet még nem használt fel a fogyasztó, továbbá a fogyasztó ne vehessen ki adatot a pufferből, ha az üres.

Ezen másodlagos vonatkozások mellett még más vonatkozások is megjelennek a rendszerben, mint a minőségi és a politikai vonatkozások. Általában ezek rendszervonatkozások – a rendszer egészére vonatkoznak, nem egyedi követelményekre vagy az azokat realizáló programokra. Ezeket *keresztvező vonatkozások*-nak nevezzük, hogy megkülönböztessük őket a *magvonatkozások*tól. A másodlagos vonatkozások is lehetnek keresztvező vonatkozások annak ellenére, hogy azok nem mindig keresztvezik az egész rendszert; inkább azokkal a magvonatkozásokkal hozhatók összefüggésbe, amelyek a kapcsolódó funkcionalitást nyújtják.

Ez látható a 32.1. ábrán, amely egy internetes banki rendszer példáját szemlélteti. Ennek a rendszernek az új ügyfelekre vonatkozó követelményei a hitelellenőrzés és a cím megerősítése. Vannak emellett a létező ügyfelek kezelésével és az ügyfelek számláinak menedzselésével kapcsolatos követelmények is. Ezek mind magvonatkozások, mivel mind a rendszer elsődleges feladatahoz tartoznak, ami egy internetes banki szolgáltatás nyújtása. Azonban léteznek a rendszernek a bank biztonsági politikáján alapuló biztonsági követelményei, valamint az esetleges rendszerhibák következtében elveszett adatok visszaállítására vonatkozó követelményei is. Ezek keresztvező vonatkozások, mivel befolyásolhatják az összes többi rendszerkövetelmény implementációját.



32.1. ábra. Keresztvező vonatkozások

A magvonatkozások szervezésére és strukturálására általában olyan programnyelvi absztrakciókat használunk, mint az eljárások és az osztályok. Azonban a magvonatkozások implementációja gyakran további kódrészleteket tartalmaz, amelyek a keresztező funkcionális, minőségi és politikai vonatkozások megvalósításához tartoznak. Ez két nemkívánatos jelenséghez vezet: keveredéshez és szétszóródáshoz.

Keveredésnek nevezzük azt, amikor a rendszer egy modulja olyan kódot tartalmaz, amely több különböző rendszerkövetelményt is megvalósít. Ezt a jelenséget illusztrálja a 32.2. ábrán látható példa, amely része a 15.14. ábrán szereplő korlátozott pufferű rendszer kódjának. Az elsődleges vonatkozást támogató kódot (ebben az esetben ez egy rekord elhelyezése a pufferben) a szinkronizációt implementáló kód teszi zavarossá. A kölcsönös kizárást (amely másodlagos vonatkozás) biztosító szinkronizációs kódot minden olyan módszerben el kell helyezni, amely el kívánja érni a megosztott tárolót. A szinkronizációért felelős kód a 32.2. ábrán az árnyékolt kódrészlet.

```
synchronized void put (SensorRecord rec) throws InterruptedException
{
    if (numberOfEntries == bufsize)
        wait ();
    store [back] = new SensorRecord (rec.sensorId, rec.sensorVal);
    back = back + 1;
    if (back == bufsize)
        back = 0;
    numberOfEntries = numberOfEntries + 1;
    notify ();
} // put
```

32.2. ábra. A pufferkezelési és szinkronizációs kód keveredése

A szétszóródás jelensége akkor keletkezik, amikor egy vonatkozás (egy logikai követelmény vagy egy követelménycsoport) implementációja szétoszlik a program több komponense között. Ez gyakran bekövetkezik, ha a másodlagos funkcionális vonatkozásokat vagy a politikai vonatkozásokat már megvalósították.

Képzeliük el például, hogy egy kórházi nyilvántartórendszernek néhány komponense felelős a személyes információk, a gyógyszerek, a konzultációk, az orvosi leletek, a diagnózisok és a gyógykezelések menedzseléséért. Ezeket különböző típusú klinikák esetén különbözőképpen konfigurálhatjuk, ezek a komponensek valósítják meg a rendszer fő vonatkozását – a páciensek adatainak kezelését.

Azonban tegyük fel, hogy a rendszernek egy igen fontos másodlagos vonatkozása a statisztikai információk kezelése – a kórház részletes információkkal szeretne rendelkezni a havonta felvett és elbocsátott páciensek számáról, arról, hogy hány beteg halt meg, milyen gyógykezeléseket hajtottak végre, milyen okokból tartottak konzultációkat stb. Ezen követelmények implementálásához olyan kódot kell adni a rendszerhez, amely névtelenné teszi az adatokat (a pá-

Patient	Image	Consultation
<attribute decls>	<attribute decls>	<attribute decls>
getName () editName () getAddress () editAddress () ... anonymise () ...	getModality () archive () getDate () editDate () ... saveDiagnosis () saveType () ...	makeAppoint () cancelAppoint () assignNurse () bookEquip () ... anonymise () saveConsult () ...

32.3. ábra. A másodlagos vonatkozásokat megvalósító módszerek szétszóródása

ciensek személyi jogainak figyelembevétele miatt), majd egy statisztikai adatbázisba írja azokat. Egy statisztikai komponens feldolgozza a statisztikai adatokat, majd előállítja a szükséges kimutatásokat.

Ezt szemlélteti a 32.3. ábra. Az ábrán három olyan osztály szerepel, amelyek megjelenhetnek a kórházi nyilvántartó rendszerben, és a páciensekkel kapcsolatos információk kezelésére szolgáló metódusokat tartalmaznak. Az árnyékolt kódrészletek azok, amelyek szükségesek a másodlagos statisztikai vonatkozások megvalósításához. Láthatjuk, hogy ez a statisztikai vonatkozás szétszóródik a magvonatkozások implementációjában.

A keveredéssel és a szétszóródással kapcsolatos problémák akkor jelentkeznek, amikor a kezdeti rendszerkövetelmények megváltoznak. Például tegyük fel, hogy a páciensek adatait kezelő rendszernek új statisztikai adatokat kell begyűjtenie. A rendszert érintő változások nem koncentrálnak egyetlen helyre, így időnk egy részét a rendszer változtatásra szoruló komponenseinek megkeresésével kell töltenünk. Ez után a megtalált komponensek mindegyikét módosítanunk kell, hogy a szükséges változtatásokat véghezvigyük. Ez költséges folyamat, mivel időt kell rá fordítanunk. Ráadásul mindig előfordulhat, hogy hibát vétünk a kód módosítása során, és néhány statisztika helytelen lesz. Annak a valószínűsége pedig, hogy hibázunk, csak növekszik a végrehajtandó változtatások számával.

32.2. ASPEKTUSOK, CSATLAKOZÁSI PONTOK ÉS VÁGÁSI PONTOK

Ebben az alfejezetben bemutatom az aspektusorientált szoftverfejlesztéssel kapcsolatos legfontosabb új fogalmakat, és szemléltetem őket a kórházi nyilvántartó rendszerből vett példákkal. Az általam használt terminológiát először az AspectJ fejlesztői vezették be az 1990-es évek végén. Ennek ellenére ezek a fogalmak általánosan használhatók és nem kötődnek egyetlen programozási nyelvhez sem. A 32.4. ábrán összefoglaltam a legfontosabb fogalmakat, amelyek megértésre szorulnak.

Kifejezés	Definíció
tanács	A vonatkozást implementáló kód.
aspektus	Egy keresztező vonatkozást definiáló programabsztrakció. Tartalmazza egy vágási pont és az aspektushoz tartozó tanács definícióját.
csatlakozási pont	Egy végrehajtás alatt álló program valamilyen eseménye, amelynek hatására az aspektushoz tartozó tanácsnak le kell futnia.
csatlakozási pont modell	A vágási pontokban hivatkozható események halmaza.
vágási pont	Egy aspektusban szereplő utasítás, amely definiálja azt a csatlakozási pontot a programban, ahol az aspektus tanácsának végre kell hajtódnia.
szöveg	Az a folyamat, amely során egy aspektusszöveg a tanácsok kódját beépíti a programkódba a megadott csatlakozási pontokban.

32.4. ábra. Az aspektusorientált szoftverfejlesztésben használt terminológia

A 32.1. alfejezetben bemutatott kórházi információs rendszer számos különböző komponenst tartalmaz, amelyek a páciensekkel kapcsolatos logikailag összefüggő információkat kezelnek. A páciens komponens a páciens személyes információit kezeli, a gyógyszer komponens a felírandó gyógyszereket tartja nyilván stb. Ha a rendszert komponensalapú szemléletben tervezzük meg, akkor a rendszer különféleképpen konfigurálható. Például készíthetünk egy konfigurációt minden különböző típusú klinika számára, így korlátozva az orvosokat arra, hogy csak az adott típusú klinikában megengedett gyógyszereket írjanak fel. Ez nagyban leegyszerűsíti a kórházi személyzet dolgát, valamint csökkenti annak az esélyét, hogy az orvos véletlenül rossz gyógyszert ír föl.

Azonban a rendszer ilyen szerkezetével az jár, hogy az adatbázis információinak frissítése több helyről történhet. Például a páciensek információi módosulhatnak a személyes adatok megváltozásakor, akkor, amikor az ajánlott gyógyszerek változnak, akkor, amikor új specialistához rendelik őket stb. Az egyszerűség kedvéért tegyük fel, hogy a rendszer komponensei konzisztens elnevezési stratégiát használnak, és az adatbázis frissítését végző metódusok mindegyike az „update” szóval kezdődik. Így a rendszerben előfordulhatnak az alábbiakhoz hasonló metódusok:

```
updatePersonalInformation (patientId, infoupdate)
updateMedication (patientId, medicationupdate)
```

A páciens a patientId azonosítja, a változtatások pedig a második paraméterben vannak kódolva – ennek részletei most számunkra nem lényegesek. A frissítéseket a kórházi személyzet tagjai hajtják végre, miután bejelentkeztek a rendszerbe.

Képzeliük el, hogy egy biztonsági rés keletkezett a rendszerben, és bizonyos információkat (a páciensekkel kapcsolatban) valaki szándékosan elrontott. Pél-

dául egy dolgozó nem jelentkezett ki, mielőtt elhagyta számítógépét, így egy illetéktelen személy hozzáférhetett a rendszerhez. Vagy akár az is megtörténhet, hogy egy belépésre jogosult személy változtatja meg ártó szándékkal az adatokat. Hogy csökkentse az eset megismétlődésének valószínűségét, a kórház egy új biztonsági intézkedést vezet be. Mielőtt a páciensek adatbázisán bárki bármilyen módosítást hajtana végre, azelőtt újra azonosítania kell magát, a változtatást igénylő személlyel kapcsolatos részletek pedig naplózásra kerülnek egy külön állományban. Ez segít nyomon követni a problémákat, ha újra bekövetkeznek.

Az új biztonsági intézkedés megvalósításának egyik módja, hogy minden komponensben módosítjuk az update módszert úgy, hogy az meghívjon más módszereket, amelyek elvégzik az azonosítást és a naplózást. Egy másik járható út az, ha a frissítő módszerek meghívása elé és mögé újabb módszerhívásokat szúrunk be, amelyek elvégzik az azonosítást és a véghezvitt változtatások naplózását. Azonban a fentiek közül egyik sem igazán jó megoldás a problémára:

1. Az első megoldás keveredő implementációt eredményez. Az adatbázis frissítése, a frissítés kezdeményezőjének azonosítása és a részletek naplózása logikailag különálló, egymástól független vonatkozások. A rendszer bármely más részébe iktathatunk be azonosítást anélkül, hogy naplózni szeretnénk, és naplózhatunk anélkül, hogy figyelemmel kísérnénk a frissítési tevékenységet. Ugyanazt az azonosító és naplózó kódrészletet több különböző módszerbe kell beszúrunk.
2. Az alternatív megközelítés szétszóródott implementációhoz vezet. Ha módszerhívásokat helyezünk el az azonosítás és a naplózás elvégzésére a frissítő módszerek minden meghívása elé és mögé, azt eredményezi, hogy a kód eloszlik a rendszer különböző részei között.

A felhasználói azonosítás és a naplózás keresztezi a rendszer magvonatkozásait, így sok különböző helyen kellene szerepelnie. Egy aspektusorientált rendszerben különálló aspektusként reprezentálhatjuk ezeket a keresztező vonatkozásokat. Egy aspektus tartalmazza annak specifikációját, hogy a keresztező vonatkozást hol kell beleszöni a rendszerbe és a kódba. Ezt szemlélteti a 32.5. ábra, amelyen az azonosítást végrehajtó aspektus definíciója látható. A példában használt jelölésmód az AspectJ szintaxisát követi, ám annak egyszerűsített változata, így megérthető az AspectJ és a Java különösebb ismerete nélkül.

Amiben az aspektusok teljesen elütnek más programnyelvi absztrakcióktól, hogy az aspektus tartalmazza annak specifikációját, hol kell őt végrehajtani. Más absztrakciók esetén (például metódusok) az absztrakció definíciója egyértelműen el van választva a használat leírásától. Egy módszer, hatáskörén belül, bárholonnan meghívható, és ránézésre nem tudjuk megmondani róla, honnan lesz meghívva. Ezzel szemben az aspektusok tartalmaznak egy vágási pontot – egy olyan utasítást, amely meghatározza, hogy az aspektust hol kell beleszöni a rendszerbe.

Ebben a példában a vágási pont egy egyszerű utasítás:

before: call (public void update* (...))

```

aspect authentication
{
  before: call (public void update* (..)) // ez egy vágási pont
  {
    // ez az a tanács, amit végre kell hajtani a szöveg után
    int tries = 0 ;
    string userPassword = Password.Get (tries) ;
    while (tries < 3 && userPassword != thisUser.password ( ))
    {
      // próbálkozást enged a jelszó megválasztására
      tries = tries + 1 ;
      userPassword = Password.Get (tries) ;
    }
    if (userPassword != thisUser.password ( )) then
      // ha a jelszó hamis, feltételezi, hogy a felhasználó elfelejtette azt
      System.Logout (thisUser.uid) ;
  }
} // authentication

```

32.5. ábra. Egy azonosító aspektus definíciója

Ez a következőt jelenti: minden olyan módszer végrehajtása előtt, amely neve az update sztringgel kezdődik (a * karaktert helyettesítő karakternek nevezzük, és minden olyan karakterláncra illeszkedik, amely azonosítókban megengedett karaktereket tartalmaz), végre kell hajtani a vágási pont definíciója után található kódot. Ez a kód a tanács, ez implementálja a keresztező vonatkozást. A mi esetünkben bekér egy jelszót a változtatást igénylő személytől, majd ellenőrzi, hogy az megegyezik-e a bejelentkezett felhasználó jelszavával. Ha nem, akkor a felhasználót kilépteti a rendszerből, és a módosítások végrehajtása nem történik meg.

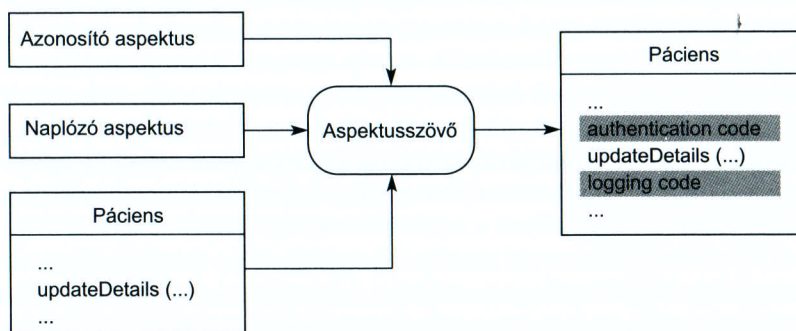
Az aspektusok egyedülálló jellegzetessége az, hogy vágási pontok használatával megadhatjuk bennük, kódjuknak hol kell végrehajtódnia. Azonban ahhoz, hogy igazán megérthessük a vágási pontok használatát, meg kell ismerkednünk a *csatlakozási pontok* fogalmával. A csatlakozási pont egy olyan esemény, amely a program végrehajtása során következik be – így csatlakozási pont lehet egy módszerhívás, egy változó inicializációja, egy mező frissítése stb. Nyilvánvalóan a programvégrehajtás során sok különböző típusú esemény következhet be. A *csatlakozási pont modell* határozza meg azokat az eseményeket, amelyekre egy aspektusorientált programozó hivatkozhat. Az AspectJ esetében például a következő események jelennek meg a csatlakozási pont modellben:

- hívási események – módszer- és konstruktorhívás;
- végrehajtási események – módszer- és konstruktorvégrehajtás;
- inicializációs események – osztály vagy objektum inicializálása;
- adatesemények – mező elérése és frissítése;
- kivételes események – egy kivétel kezelése.

A vágási pont definiálja azokat az eseményeket (például egy adott nevű eljárás meghívása), amelyekhez a tanácsot hozzá kell rendelni. Ez azt jelenti, hogy egy tanács a programba számos helyen beleszőhető. Például:

1. Egy adott módszer, adott nevű módszerek listája vagy egy adott mintára illeszkedő nevű módszerek meghívása elé.
2. Egy módszerből való normális vagy kivételes visszatérés után. A 3.25. ábrán látható példa esetén definiálhatunk egy vágási pontot, amely végrehajtja a naplózási kódot a frissítő módszerek meghívása után.
3. Ha egy objektum valamely mezője módosul, egy tanács segítségünkre lehet a módosítás monitorozásában vagy a mező megváltoztatásában.

A tanácsok beszúrása a vágási pontokban meghatározott helyekre az aspektusszövő felelőssége. Ezek a szövők a fordítók olyan kiterjesztései, amelyek feldolgozzák az aspektusokat, valamint a rendszert alkotó osztályokat és módszereket. Az összeszövés eredménye egy új program, amelybe beépültek a tanácsok a vágási pontokban definiált helyeken. Az aspektusok integrálása úgy történik, hogy a keresztező vonatkozások a végső rendszerben a megfelelő helyen hajódnak végre.



32.6. ábra. Aspektus-összeszövés

A 32.6. ábra a pácienskezelő rendszer példájában megismert azonosító és naplózási aspektusok összeszövését szemlélteti. Az aspektusok összeszövésének három különböző megközelítése létezik:

1. A forráskód előfeldolgozása. Ebben az esetben a szövő program forráskódot kap bemenetként, és ebből egy újabb forrásnyelvű állományt generál (például Java vagy C++), amely már lefordítható egy hagyományos fordítóval. Ez a szemlélet jelenik meg az AspectX nyelvben és a hozzá kapcsolódó XWeaver szövőben (Birrer és mások, 2005).
2. Összeszövés szerkesztési időben. Ebben az esetben a fordítót úgy módosítják, hogy tartalmazzon egy szövőt. Egy aspektusorientált nyelvű (AspectJ) forrás feldolgozása után szabványos Java-bájt kód keletkezik. Ez a továbbiakban vég-

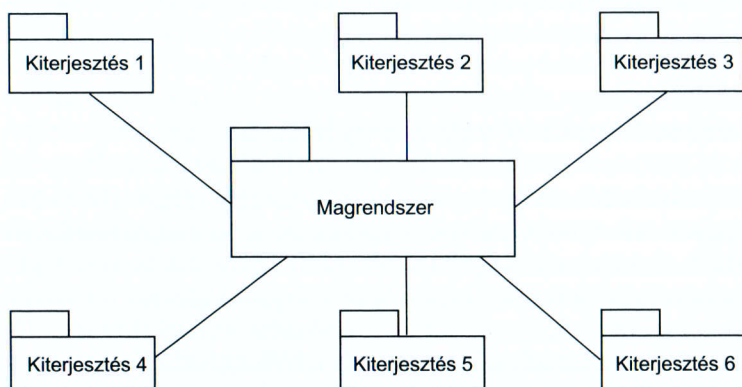
rehajtható közvetlenül egy Java-interpreter segítségével, vagy natív gépi kód állítható elő belőle. A könyv írása idején ez a legelterjedtebb megközelítés.

3. Dinamikus összeszövés végrehajtási időben. Ez a legáltalánosabb megközelítés, viszont ez okozza a legnagyobb többletterhelést. Ebben az esetben a rendszer figyeli a csatlakozási pontokat, és mikor egy olyan esemény következik be, amelyre egy vágási pont hivatkozik, a megfelelő tanács beépül a végrehajtás alatt álló programba.

32.3. SZOFTVERFEJLESZTÉS ASPEKTUSOKKAL

Az aspektusokat eredetileg programnyelvi konstrukciókként vezették be, ám ahogy korábban megjegyeztem, a vonatkozások ötlete valójában a rendszerkövetelményekből származik. Így van értelme annak, hogy az aspektusorientált szemléletmódot a szoftverfejlesztési folyamat minden állomására alkalmazzuk. A fejlesztés korai szakaszaiban ez azt jelenti, hogy a vonatkozások szétválasztásának ötletét kell alapul vennünk a követelmények meghatározása, valamint a rendszer megtervezése során. Azaz a vonatkozások azonosításának és modellezésének szerves részét kell képeznie a követelményfeltárási és rendszertervezési folyamatoknak. Az aspektusorientált programozási nyelvek ezután megfelelő technológiai támogatást biztosítanak a vonatkozások szétválasztásához a rendszer implementációjában.

Jacobsen és Ng (Jacobsen és Ng, 2004) azt javasolja, hogy tekintsük úgy a rendszereket, mint egy, a különböző megrendelők érdekeit támogató mag és a hozzá kapcsolódó kiegészítések együttese. A 32.7. ábra ezt szemlélteti. Az ábrán UML-csomagokat használtam mind a mag, mind a kiegészítések megjelenítésére. A magrendszer a rendszer alapvető céljait támogató rendszerfunkciók halmaza. Ennek alapján, ha a rendszer célja a kórház pácienseiről szóló információk karbantartása, akkor a rendszer lehetővé teszi a páciensek rekordjait tartalma-



32.7. ábra. Magrendszer kiterjesztésekkel

zó adatbázis létrehozását, szerkesztését, kezelését és elérését. A kiterjesztések a megrendelők további igényeit tükrözik, amelyeket a magrendszerrel integrálni kell. Például nagyon fontos, hogy egy kórházi információs rendszer bizalmasan kezelje a páciensek információit, így egy lehetséges kiterjesztés feladata lehet a hozzáférés ellenőrzése, a titkosítás stb.

Sok különböző típusú kiterjesztés létezik, amelyek mind a 32.1. alfejezetben tárgyalt vonatkozástípusokból származnak:

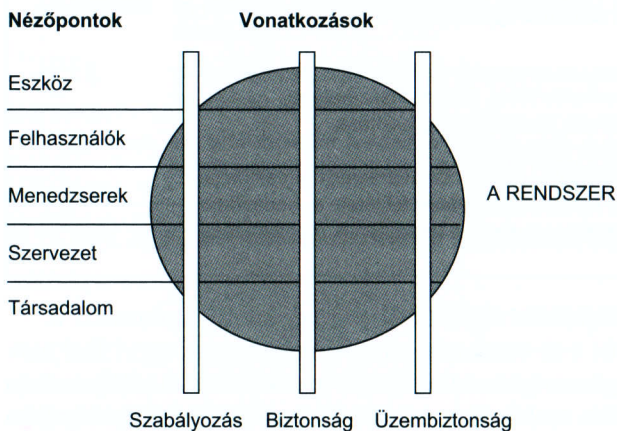
1. *Másodlagos funkcionalitással kapcsolatos kiterjesztések.* Ezek további funkcionalitást adnak a magrendszer által nyújtott lehetőségekhez. Például kórházi információs rendszer esetén ilyen kiegészítés lehet az a funkció, amely lehetővé teszi az előző hónapban felírt gyógyszerekről szóló adatok lekérdezését.
2. *Politikai kiterjesztések.* Ezek a szervezet politikáját támogató funkciókat adnak a rendszerhez. A biztonsági eszközöket megvalósító kiterjesztések kitűnő példát szolgáltatnak a kiterjesztések ezen típusára.
3. *Szolgáltatások minőségével kapcsolatos kiterjesztések.* A rendszer szolgáltatásainak minőségével kapcsolatos követelmények támogatásához szükséges funkciókat adnak a rendszerhez. Idetartozik például az a kiterjesztés, amely gyorsított támogatást nyújt az adatbázis-elérések számának csökkentésére, valamint az, amely biztosítja a rendszerhibák esetén szükséges helyreállítások végrehajtásához az automatikus biztonsági mentést.
4. *Infrastrukturális kiterjesztések.* Olyan funkcionális lehetőségeket adnak a rendszerhez, amelyek segítik a rendszer megvalósítását egy adott platformon. A kórházi információs rendszer példájánál maradva, infrastrukturális kiterjesztések használhatók az adatbázishoz kapcsolódó interfész megvalósításához. Ha az adatbázist lecseréljük, csak az infrastrukturális kiterjesztések módosítására van szükség.

A kiterjesztések mindig egyfajta funkcionalitást vagy eszközrendszert adnak a magrendszerhez. Az aspektusok használata az egyik módszer ezen kiterjesztések implementációjához. Az aspektusok összekapcsolása a magrendszer funkcionalitásával az aspektusorientált programozási környezet szövőjének segítségével történik.

32.3.1. Vonatkozásorientált követelményfeltárás

Ahogy az a 32.1. alfejezetben említettem, a vonatkozások a megrendelők követelményeit tükrözik. Ezek a vonatkozások vonatkozhatnak a megrendelő által elvárt funkcionalitásra, a rendszerszolgáltatások minőségére, a szervezeti politikára, valamint a rendszer egészével kapcsolatos tulajdonságokra. Tehát van értelme a megrendelői igények azonosítását és specifikálását megcélzó szemléletmód alkalmazásának a követelmények feltárásában. Néha a *korai aspektusok* kifejezéssel hivatkozunk az aspektusoknak a szoftveréletről korai szakaszaiban történő alkalmazására, ahol a hangsúly a vonatkozások szétválasztásán van.

A vonatkozások szétválasztásának fontosságát a követelmények feltárása során már sok éve felismerték. A nézőpontok, amelyeket a 7.2.1. alfejezetben tárgyaltam, és amelyeket számos követelményfeltárási módszerbe beépítettek (Finkelstein és mások, 1992; Easterbrook és Nuseibeh, 1996; Kotonya és Sommerville, 1996), alkalmas eszközt nyújtanak a különböző megrendelők által elvárt vonatkozások szétválasztására. Ezek a különböző megrendelői csoportok egymástól elkülöníthető funkcionális igényeit tükrözik. Azonban vannak olyan követelmények is, amelyek az összes nézőpontot keresztezik. Ez látható a 32.8. ábrán. Az ábra azt szemlélteti, hogy a nézőpontok lehetnek különböző típusúak, de a keresztező vonatkozások (szabályozás, megbízhatóság, biztonság stb.) olyan követelményeket generálnak, amelyek hatással vannak minden nézőpontra. Ez volt a fő irányelv a PreView-módszer (Sommerville és Sawyer, 1997; Sommerville és mások, 1998) fejlesztése alatt, amely tartalmazott olyan lépéseket, amelyek a keresztező, nemfunkcionális vonatkozások azonosítását szolgálták.



32.8. ábra. Nézőpontok és vonatkozások

Ahhoz, hogy a 32.7. ábrához hasonlóan szervezett rendszert fejleszthessünk, azonosítanunk kell a magrendszer követelményeit, valamint a kiegészítésekkel kapcsolatos követelményeket. A követelményfeltárási nézőpont-orientált megközelítése, ahol minden nézőpont a megrendelők egy csoportjának követelményeit reprezentálja, egy jó módszer a mag és a másodlagos vonatkozások szétválasztásához. Ha a megrendelői nézőpontok mentén szervezzük a követelményeket, akkor könnyen elemezhetjük őket, így egyszerűen felfedezhetjük azokat az összefüggő követelményeket, amelyek megjelennek minden, vagy majdnem minden nézőpontban. Ezek adják a rendszer magfunkcionalitását. A többi követelmény a különálló vonatkozásokra vonatkozik, amelyek a magfunkcionalitás kiterjesztéseként valósíthatók meg.

A 32.9. ábrán egy olyan rendszer követelményeinek vázlata szerepel három lehetséges nézőpont szerint csoportosítva, amely a katasztrófaszolgálatok által használt speciális eszközök nyilvántartását végzi. Ezek az eszközök a régió vagy

állam különböző pontjain vannak elhelyezve, és ha valamilyen katasztrófa következik be, a sürgősségi szolgálatok ezt a rendszert használják a vészhelyezethez minél közelebbi helyen rendelkezésre álló eszközök megkeresésére.

1. A sürgősségi szolgáltatás felhasználói

- 1.1. Egy bizonyos típusú eszköz megtalálása (például daru)
- 1.2. Egy raktárban található eszközök megtekintése
- 1.3. Eszköz kikölcsönzése
- 1.4. Eszköz visszaadása
- 1.5. A katasztrófa helyszínére szállított eszköz felállítása
- 1.6. Kár jelentése
- 1.7. A vészhelyezethez közeli raktárak megtalálása

2. Katasztrófaszolgálatok

- 2.1. Egy bizonyos típusú eszköz megtalálása
- 2.2. Egy raktárban található eszközök megtekintése
- 2.3. Eszköz eltávolítása és felvétele
- 2.4. Eszköz áthelyezése egy raktárból egy másikba
- 2.5. Új eszköz rendelése

3. Karbantartó személyzet

- 3.1. Eszköz kikölcsönzése karbantartásra, eszköz visszaadása
- 3.2. Az összes raktárban található eszközök megtekintése
- 3.3. Egy bizonyos típusú eszköz megtalálása
- 3.4. Egy adott eszközre vonatkozó karbantartási ütemezés megtekintése
- 3.5. Egy eszközre vonatkozó karbantartási rekord felvétele a rendszerbe
- 3.6. Egy raktár minden karbantartásra szoruló eszközének megtekintése

32.9. ábra. Egy eszköz-nyilvántartási rendszer nézőpontjai

A példában láthatjuk, hogy a megrendelők által képviselt összes nézőpontban megjelenik az igény a speciális eszközök megkeresésére, a különböző helyeken rendelkezésre álló szabad eszközök közötti böngészésre, valamint egy eszköz kikölcsönzésére és visszahelyezésére. Tehát ezek a magrendszer követelményei. A másodlagos követelmények a különböző nézőpontok specifikus igényeinek kielégítését szolgálják. A másodlagos követelmények olyan kiegészítések létrehozását teszik szükségessé, amelyek támogatják az eszközhasználatot, az eszközkezelést és -karbantartást.

A nézőpontok használatával felismert másodlagos vonatkozások nem kereszteznek szükségszerűen minden nézőpontot. Ezek inkább a vonatkozások szétválasztása szempontjából jelentősek. Emellett azonban léteznek olyan keresztező vonatkozások, amelyek minden nézőpont kapcsán fontosak. Ezek gyakran a politikai és a szolgáltatásminőségi követelményeket tükrözik, amelyek a rendszer egészére vonatkoznak. Ilyen keresztező vonatkozások előállhatnak olyan egyedi nézőpont-követelményekből, amelyek minden nézőpont szempontjából jelentőséggel bírnak, illetve megjelenhetnek a rendszer átfogó tulajdonságainak tárgyalása során is.

Az eszköz-nyilvántartási rendszer példájában ilyen keresztező vonatkozás a rendszer megfelelő rendelkezésre állásának szükségessége. A katasztrófák bekö-

vetkeztét ritkán előzi meg bármilyen előjel. Az életek megmentése gyakran az alapvető eszközök mihamarabbi alkalmazását követeli meg. Emiatt az eszköz-nyilvántartási rendszerrel szemben támasztott egyik legfontosabb követelménynek a magas fokú rendelkezésre állásnak kell lennie. A 32.10. ábrán néhány ilyen követelmény látható, a hozzájuk tartozó ésszerű magyarázatokkal együtt. Ezekből a követelményekből könnyen megállapítható, hogy a magrendszerhez szükségesek olyan kiegészítések, amelyek a tranzakciók naplózásáért és az állapotok lekérdezéséért felelősek. Ezek egyszerűbbé teszik a problémák felismerését és az esetleges átváltást a rendszer egy korábbi állapotába.

AV.1.	Léteznie kell egy készenléti rendszernek a fő rendszertől távoli helyen.
<i>Magyarázat:</i>	A katasztrófa hatással lehet a fő rendszer helyére is.
AV.1.1.	Minden tranzakciót naplózni kell mind a fő, mind a készenléti rendszer oldalán.
<i>Magyarázat:</i>	Így lehetővé válik a tranzakciók visszajátszása, amivel a rendszer konzisztenssé tehető.
AV.1.2.	A rendszernek ötpercenként állapotinformációkat kell küldenie a katasztrófavédelem irodájába.
<i>Magyarázat:</i>	Ha a fő rendszer használhatatlanná válik, a katasztrófavédelemnek tudomást kell szereznie erről, hogy a készenléti rendszerre válthasson.

32.10. ábra. Az eszköz-nyilvántartási rendszer rendelkezésre állással kapcsolatos követelményei

A követelményfeltárási folyamat végeredménye egy olyan követelményhalmozat kell hogy legyen, amely a magrendszer és a kiterjesztések fogalma mentén szerveződik. Például az eszköz-nyilvántartási rendszerben a magkövetelmények a következők lehetnek:

- M.1 A rendszernek lehetővé kell tennie bármely arra jogosult felhasználó számára, hogy megtekintse bármely eszköz leírását a nyilvántartásban.
- M.2 A rendszer tartalmazhat keresési lehetőséget, amely segítségével a jogosultsággal rendelkező felhasználók az egyedi raktárakban vagy minden raktárban egyszerre kereséseket hajthatnak végre. A keresés végrehajtható egy bizonyos készülékre és egy adott készüléktípusra is.

A rendszer tartalmazhat olyan kiegészítéseket, amelyek támogatják az eszközök beszerzését és áthelyezését. Ezen kiterjesztés követelményei az alábbiak lehetnek:

- K1.1 A rendszer tegye lehetővé a jogosult felhasználók számára, hogy egy adott eszköz cseréjét kérjék akkreditált szolgáltatóknak történő rendelés leadásán keresztül.
- K1.1.1 Amikor valaki valamilyen eszközt rendel, akkor azt az eszközt el kell helyezni valamelyik raktárban, és megjelölni a „rendelés alatt” címkével.

Ezen a szinten a legfontosabb az, hogy ne határozzunk meg túl sok vonatkozást és kiterjesztést. Ez csak összezavarja az olvasót, és elhamarkodott tervezési döntésekhez vezet. Az ilyen döntések azután korlátozzák a tervezők szabadságát, és egy olyan rendszert eredményeznek, amely nem felel meg a minőségi követelményeknek.

32.3.2. Aspektusorientált tervezés és programozás

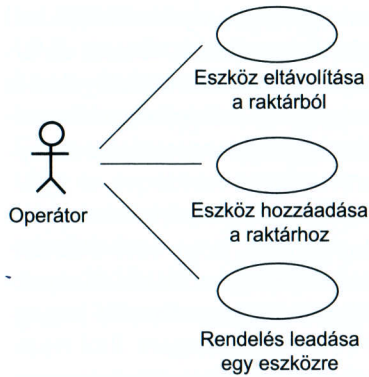
Az aspektusorientált tervezés a rendszer tervezésének az a folyamata, amely a követelmény-feltárás során meghatározott keresztező vonatkozások és kiterjesztések megvalósításához az aspektusokat hívja segítségül. Ezen a ponton a cél a megoldandó problémához tartozó vonatkozások lefordítása a megoldás implementációjában részt vevő aspektusokká. Emellett meg kell értenünk azt, hogy a különálló komponensek hogyan alakítják majd ki a teljes rendszert, valamint biztosítanunk kell, hogy nem lép fel semmiféle kétértelműség a kompozíció során.

A követelmények magas szinten történő feltárása kitűnő alapot szolgáltat annak felismeréséhez, hogy mely rendszerkiterjesztéseket érdemes aspektusokként implementálni. Ezután ezeket a kiterjesztéseket tovább kell részleteznünk esetleges további kiterjesztések azonosításához, továbbá az elvárt funkcionalitás megértéséhez. Az egyik módja ennek néhány felhasználói eset azonosítása (ezt a 7. fejezetben tárgyaltam) minden nézőpont esetén. A használati esetek átjárót képeznek a követelmények és a tervezés között. Sokkal részletesebbek, mint a követelmények leírása. Egy használati eset diagramon megjelennek a felhasználói interakciók lépései, így megkezdjük a rendszer osztályainak azonosítását.

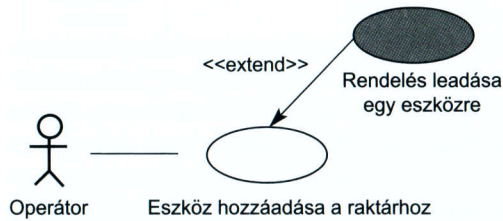
Jacobsen és Ng (Jacobsen és Ng, 2004) írtak egy könyvet arról, hogy a használati esetek hogyan játszhatnak alapvető szerepet az aspektusorientált szoftverfejlesztésben. Minden részletre kiterjedően tárgyalják (ami meghaladja jelen könyv kereteit), hogy a használati eset szemléletű megközelítés hogyan szolgálhat az aspektusorientált szoftverfejlesztés alapjául. Azt javasolják, hogy minden aspektusnak feleltessünk meg egy használati esetet, és olyan kiegészítéseket ajánlanak a használati eset alapú megközelítéshez, amelyek támogatják a vágási pontokat és a csatlakozási pontokat. Bevezetik továbbá a használati eset szeletek és a használati eset modulok fogalmát, amelyek olyan osztálytöredékeket tartalmaznak, amelyek kompozíciójából létrejöhet a teljes rendszer.

A 32.11. ábrán három olyan használati eset látható, amely szerepelhet az eszköz-nyilvántartási rendszerben. Ezek az eszköz leltárba vételének és a rendelésnek felelnek meg. Az eszközrendelés és egy eszköz leltárba vétele egymással összefüggő vonatkozások. Mihelyt egy megrendelt eszköz megérkezett, azt leltárba kell venni, majd el kell helyezni egy raktárban.

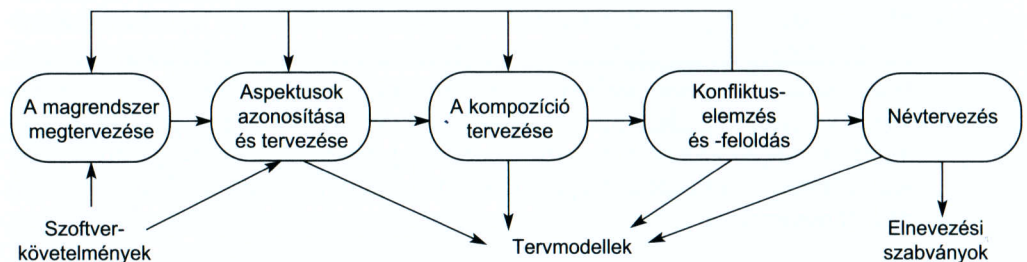
Az UML jelenleg is tartalmazza a használati eset kiterjesztés fogalmát, amit akkor használunk, ha egy használati eset egy másik használati eset funkcionalitását terjeszti ki. A 32.12. ábrán az látható, hogyan terjeszti ki az eszközrendelés az eszköz elhelyezését egy raktárban. Ha az elhelyezendő eszköz nem létezik, akkor lehetőség van a megrendelésre. Ebben az esetben az eszköz azután kerül a rak-



32.11. ábra. Használati esetek az eszköznyilvántartó rendszerből



32.12. ábra. Kiterjesztés használati esetek



32.13. ábra. Az aspektusorientált tervezés általános folyamata

tárba, miután az megérkezett. A használati eset modellek fejlesztése során meg kell figyelniük, mik a modellek közös jellemvonásai, és ahol lehetséges, ki kell emelniük egy magot, amely köré kiterjesztésként szervezzük a maradék funkcionálisit. A keresztező vonatkozások is, mint a tranzakciók naplózása, reprezentálhatók kiterjesztésekként. Jacobsen és Ng azt is tárgyalja, hogy az ilyen típusú kiterjesztések hogyan implementálhatók aspektusként.

Ha aspektusorientált tervezés használatára készülünk, akkor elengedhetetlen egy hatékony folyamat kifejlesztése. Véleményem szerint az aspektusorientált tervezési folyamatnak tartalmaznia kell a 32.13. ábrán látható tevékenységeket. Ezek a következők:

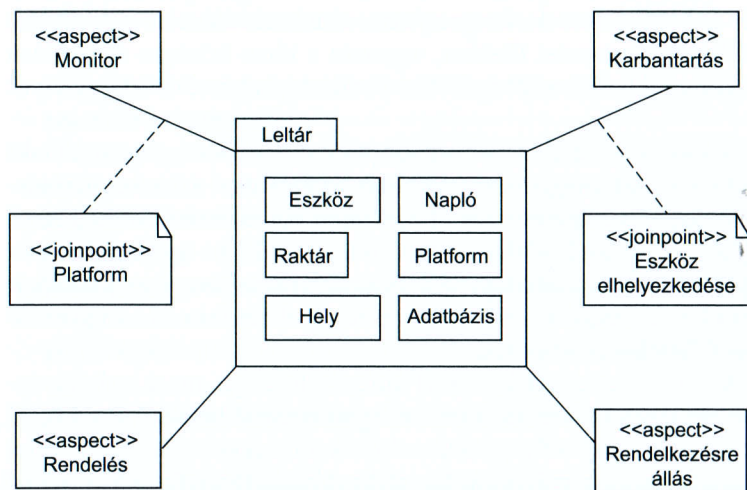
1. *A magrendszer megtervezése.* Ebben a szakaszban egy olyan architektúrát kell terveznünk, amely támogatja a rendszer magfunktionalitását. Ennek az architektúrának a szolgáltatások minőségével kapcsolatos követelményeket is figyelembe kell vennie, mint például a hatékonysági és a függőségi követelmények. Ugyanakkor a szükséges specifikus támogatás kiterjesztésként implementálható.
2. *Aspektusok azonosítása és tervezése.* A rendszerkövetelmények feltárása során felismert kiterjesztéseket elemezve meg kell állapítanunk, hogy ezek már maguk is aspektusok, vagy néhányat aspektusokra kell bontanunk. Ha az aspektusokat azonosítottuk, azok már egymástól függetlenül tervezhetők, a magrendszer eszközeinek figyelembe vételével.
3. *A kompozíció tervezése.* Ezen a ponton elemeznünk kell a magrendszer és az aspektusok terveit, és meg kell vizsgálnunk, hogy az aspektusok hogyan illeszthetők a magrendszerhez. Azaz itt azonosítjuk a program azon csatlakozási pontjait, ahová az aspektusok be lesznek szőve.
4. *Konfliktuselemzés és -feloldás.* Az egyik probléma, amely az aspektusokkal kapcsolatban felmerülhet az, hogy az aspektusok befolyásolhatják egymás működését, mikor hozzáillesztjük őket a rendszerhez. Konfliktus akkor lép fel, ha több különböző aspektusnak a specifikációjuk alapján a program ugyanazon pontján kellene a rendszerbe beépülnie. Azonban léteznek ennél szövevényesebb konfliktusok is. Amikor az aspektusokat egymástól függetlenül fejlesztik, azok feltételezésekkel élhetnek a kiterjesztendő magrendszer funktionalitását illetően. Azonban az aspektusok összeillesztése során a magrendszer funktionalitása úgy módosulhat, hogy a kezdeti feltevések érvénytelenné válnak.
5. *Névtervezés.* Ez egy fontos tervezési tevékenység. Ebben a lépésben szabványokat definiálunk a program entitásainak elnevezésére. Ennek alapvető szerepe van az akaratlan vágási pontok problémájának elkerülésében. A probléma akkor jelentkezik, amikor egy név illeszkedik egy vágási pont mintájára anélkül, hogy ezt valaki szándékosan így tervezte volna. Ha az elnevezési szabványok kifejlesztése megtörtént, a modellek elemeit ennek megfelelően át kell neveznünk.

Ez természetesen interaktív folyamat, amelynek elején kezdeti indítványokat tehetünk, majd később, a tervezés menetének elemzése és mélyebb megértése után, ezeket újrafogalmazhatjuk. Általában a követelmények között felismert kiterjesztések felülvizsgálata elengedhetetlen, ugyanis valószínűleg nagyobb számú aspektusra van szükség a rendszerben.

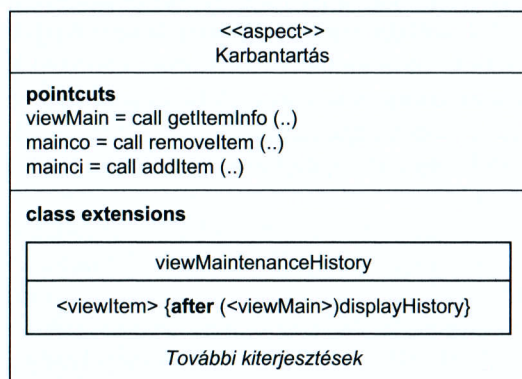
Az aspektusorientált tervezési folyamat végeredménye egy aspektusorientált modell. Ez az UML egy kiterjesztett változatával szemléltethető, amely tartalmaz aspektusspecifikus konstrukciókat, például amelyeket Clarke és Baniassad (Clarke és Baniassad, 2005) vagy Jacobsen és Ng (Jacobsen és Ng, 2004) javasoltak. Az „aspektus UML” alapvető elemei között vannak olyan eszközök, amelyek az aspektusok modellezését, és olyanok, amelyek a csatlakozási pontoknak és a tanácsok beszúrási helyének specifikálását támogatják. Azonban még nem született szabvány az aspektusok UML-be foglalására.

A 32.14. ábrán egy aspektusorientált modell látható. Az aspektusok jelölésére UML sztereotípiát használtam, ahogyan Jacobsen és Ng javasolta. Az ábra az eszköz-nyilvántartási rendszer magját, és néhány, a rendszerrel összekapcsolandó aspektust tartalmaz. A diagramon feltüntettem a magrendszer néhány osztályát. Ez természetesen egy egyszerűsített ábra – a teljes modell ennél sokkal több osztályt és aspektust tartalmazna. Figyeljük meg, hogyan használtam az UML-jelöléseket az aspektusok által keresztezett osztályokról szóló információk megjelenítésére.

A 32.15. ábra egy aspektusmodell bővebb változata, amely több információt enged látnunk az aspektusról. Nyilván az aspektusok tervezése előtt a magrendszer kell megterveznünk. A hely szűkössége miatt számos feltevéssel éltem a magrendszer metódusaival és osztályaival kapcsolatban.



32.14. ábra. Egy aspektusorientált modell



32.15. ábra. Egy aspektus részmodellje

Az aspektus első szakasza felsorolja a vágási pontokat, amelyek azt specifikálják, hogy hol épül be az aspektus a magrendszerbe. Például az első vágási pont azt határozza meg, hogy az aspektust a `call getItemInfo` (...) csatlakozási pontnál kell a rendszerhez illeszteni. A következő szakasz az aspektus által megvalósított kiterjesztéseket definiálja. Jelen példában a kiterjesztési utasítás a következőképpen értelmezhető:

A `viewItem` metódusban, a `getItemInfo` metódus meghívása után, be kell szűrni a `displayHistory` metódus meghívását a nyilvántartási bejegyzés megjelenítéséhez.

Az aspektusorientált programozás a Xerox PARC laboratóriumából indult útjára 1997-ben, az AspectJ programozási nyelv kifejlesztésével. Azóta is ez a legelterjedtebb aspektusorientált nyelv, de más nyelvekhez (C#, C++) is léteznek már kiterjesztések. A vonatkozások explicit szétválasztására és kompozíciójára más kísérleti nyelveket is kifejlesztettek. Itt azonban nem bocsátkozom részletekbe az aspektusorientált programozást illetően, ugyanis a téma bőséges tárgyalása megtalálható más könyvekben (Laddad, 2003b; Gradecki és Lezeiki, 2003; Colyer és mások, 2005).

Ha rendszerünk megtervezését aspektusorientált szemlélettel végeztük, akkor már azonosítottuk annak magfunktionalitását, és az ehhez a funktionalitáshoz kapcsolódó kiterjesztéseket, amelyeket keresztvevő vonatkozásként kell implementálnunk. Így a programozási folyamat során a hangsúlyt a magrendszer és a kiterjesztések megvalósítására helyezhetjük, valamint arra, hogy megfelelően specifikáljuk az aspektusok vágási pontjait, hogy az aspektusokhoz tartozó tanácsok jó helyen szövédjének a rendszerbe.

A vágási pontok helyes specifikálása azért kulcsfontosságú, mert ez határozza meg, hogy az aspektusok hol épülnek be a magfunktionalitásba. Ha a vágási pontok specifikációja során valamilyen hibát vétünk, a tanácsok nem a megfelelő helyen fognak a programba szövédni. Ez pedig kiszámíthatatlanná és megjósolhatatlanná teheti a program viselkedését. Nagyon fontos a rendszertervezés során lefektetett elnevezési szabványok követése. Emellett át kell vizsgálnunk minden aspektust annak érdekében, hogy szavatoljuk, hogy ha több aspektust szövéünk a magrendszerbe ugyanannál a csatlakozási pontnál, azok nem fogják megzavarni egymás működését. Általában az a legjobb, ha teljesen elkerüljük az ilyen szituációkat, de olykor mégis ez az egyetlen, amit tehetünk. Ilyenkor biztosítanunk kell, hogy az aspektusok teljesen függetlenek legyenek egymástól. A program viselkedése nem függhet attól, hogy az aspektusokat milyen sorrendben szövjük a programba.

32.3.3. Verifikáció és validáció

Ahogy a 22. fejezetben tárgyaltam, verifikáció és validáció alatt azt a folyamatot értjük, amely során demonstráljuk, hogy a program megfelel a specifikációjának (verifikáció), és eleget tesz a megrendelők valódi követelményeinek (validáció). A statikus verifikációs technikák a forráskód manuális vagy automatikus

átvizsgálását helyezik előtérbe, míg a dinamikus validáció és a tesztelés célja a programbeli hiányosságok felderítése, valamint a program helyességének ellenőrzése. Ha a hiányosságok felfedezése a cél, a tesztelésben jó segítséget jelenthet a forráskód ismerete. A tesztlefedettség-metrikák a teszt hatékonyságát hivatottak kifejezni abban az értelemben, hogy mennyire hatékony a teszt a végrehajtott utasítások szempontjából.

A validáció és tesztelés folyamata az aspektusorientált rendszerek esetén sem tér el a többi rendszernél megszokottól. A programot fekete dobozként kezelik, és olyan tesztek találhatók ki, amelyek végrehajtása után kiderül, hogy a rendszer megfelel a követelményeinek vagy sem. Azonban, ha aspektusokat használunk, komoly akadályok hárulnak a programátvizsgálás elé, valamint a fehér doboz tesztelés elé, ahol a forráskódot használják a lehetséges hibaforrások azonosításához. Ráadásul, ahogyan Katz (Katz, 2005) rámutatott, az aspektusok használata további problémákkal szembesíti a programok tesztelőit:

1. Hogyan lehet úgy specifikálni az aspektusokat, hogy azokból teszteseteket származtathassunk?
2. Hogyan tesztelhetjük az aspektusokat attól az alaprendszerrel függetlenül, amelybe azok bele fognak szöveődni?
3. Hogyan tesztelhető az aspektusok interferenciamentessége? Korábban említettem, hogy az aspektusok interferenciája akkor jelentkezik, amikor két aspektus ugyanazt a csatlakozási pont specifikációt használja.
4. Hogyan tervezhetők olyan tesztek, amelyek előidéznek minden csatlakozási pont végrehajtását, és hogyan alkalmazhatók a megfelelő aspektustesztek?

A fenti problémák mindegyike további tanulmányozást igényel, és a könyv írásának idején nem kerültek nyilvánosságra általános megoldások. Ezek a problémák azért jelentkeznek, mert az aspektusok nem lazán, hanem szorosan kapcsolódnak az alaprendszerhez. Emiatt nehéz őket elkülönítve tesztelni. Továbbá, mivel a programba annak bármely részén beleszőhetők, ezért nem lehetünk biztosak abban, hogy egy aspektus, amely működik a program egy adott pontján, helyesen működik mindenhol. A tesztelés problémáit nem tárgyalom bővebben, helyettük arra helyezem a hangsúlyt, hogy az aspektusorientált megközelítés milyen nehézségeket okoz a programátvizsgálás során.

A programátvizsgálásokban, amelyekről a 22. fejezetben volt szó, egy olvasócsapat vesz részt. A csapat átolvassa a program forráskódját, hogy felfedezze a programozó által vétett hibákat. Ez egy nagyon hatékony módja a hiányosságok felfedezésének. Azonban az aspektusorientált programok nem olvashatók folytonosan felülről lefelé. Emiatt az ilyen programokat az emberek nehezebben értik meg.

A program érthetőségének általános szabálya, hogy a forráskódnak balról jobbra, felülről lefelé olvashatónak kell lennie, hogy az olvasónak ne kelljen figyelmezt a kód egyik részéről a másokra váltogatnia. Ez a szabály könnyebbé teszi az olvasók dolgát, valamint növeli annak az esélyét, hogy a programozók nem hibáznak, mivel figyelmük a program egyetlen szakaszára összpontosul. A prog-

ram olvashatóságának növelése volt a mozgatórugója többek között a strukturált programozás (Dijkstra és mások, 1972) bevezetésének és a feltétel nélküli elágaztató utasítások (goto) kihagyásának a magas szintű programozási nyelvekből.

Egy aspektusorientált rendszerben a szekvenciális kódolvasás kivitelezhetetlen. Az olvasónak át kell tekintenie minden aspektust, meg kell értenie a csatlakozási pontokat (amelyek lehetnek mintákkal adottak is), továbbá ismernie kell az aspektusorientált programozási nyelv vágási pont modelljét. Ezután a program olvasása során azonosítania kell minden lehetséges csatlakozási pontot, és figyelmét az aspektus kódjára fordítva, meg kell vizsgálnia, hogy az adott ponton az beleszűrhető-e a programba. A figyelmét ezután újra az alaprendszer kódjának vizsgálatára fordíthatja. A valóságban ez kognitív szempontból lehetetlen, így az aspektusorientált programok átvizsgálásában csak az automatikus programátvizsgálási eszközök nyújthatnak segítséget.

Ezen eszközök között lehetnek olyanok, amelyek egyetlen sík állományt készítenek az aspektusorientált programból, és olyan programot kínálnak az olvasónak, amelyben az aspektusok a megfelelő csatlakozási pontokon beleszűrve szerepelnek. Azonban néhány aspektusorientált nyelv dinamikus csatlakozási pont modellt használhat, így lehetetlen feladat annak bizonyítása, hogy a sík állománnyá alakított program pontosan ugyanúgy viselkedik, mint a dinamikusan összeszűrt változat. Ráadásul, mivel megengedett, hogy több aspektus is ugyanazt a vágási pont specifikációt használja, az eszköznek azt is tudnia kell, hogy a szövő hogyan kezeli ezeket a „versenyző” aspektusokat, és hogy a kompozíció milyen sorrendben történik.

A 23. fejezetből tudjuk, hogy a fehér doboz vagy strukturális tesztelés egy olyan szisztematikus megközelítése a tesztelésnek, amelyben a program forráskódjának ismerete segít a hiányossági tesztek összeállításában. A cél olyan tesztek kifejlesztése, amelyek valamilyen szinten lefedik a programot. Tipikusan a teszteseteknek garantálniuk kell, hogy a program minden logikai ága ellenőrzésre kerül, aminek a következménye az, hogy minden utasítás legalább egyszer végrehajtódik. Annak ellenőrzésére, hogy a program lefedése megfelelő mértékű volt, programvégrehajtás-elemzők használatosak.

Az aspektusorientált rendszerekben két probléma merül fel ezen megközelítés használata kapcsán:

1. Hogyan segíti a program forráskódjának ismerete a tesztesetek szisztematikus származtatását?
2. Mit jelent pontosan a program lefedettsége?

Egy feltétel nélküli elágaztató utasításokat nem tartalmazó, strukturált programhoz történő teszt tervezéséhez elkészíthetünk egy programfolyamat gráfot, amely felfedi a lehetséges végrehajtási utakat. Ezután csak ellenőriznünk kell a program forrását, és a bemeneti értékeket úgy kell megválasztanunk, hogy garantáltan végrehajtódjon minden ilyen út.

Azonban egy aspektusorientált program nem strukturált program. A program menetét megszakíthatják bizonyos utasítások (Constantinos és mások, 2004).

A programvégrehajtás csatlakozási pontjainál aspektusok hajtódhatnak végre. Nem világos, hogyan nézne ki egy ilyen programfolyamat gráfja. Így tehát bonyolult olyan tesztek generálni, amelyek szavatolják, hogy a kód és az aspektusok minden lehetséges kombinációja végrehajtásra kerül.

Aspektusorientált rendszerek esetén annak eldöntése is problémát jelent, hogy mi a teszt lefedettsége. Azt jelenti talán, hogy a program minden aspektusa végrehajtódik legalább egyszer? Az aspektusok működése könnyen megzavarhatja egymást, miután beépültek a programba, így ez nagyon gyenge kritérium. Vagy azt jelenti a tesztlefedettség, hogy minden aspektus kódjának legalább egyszer végre kell hajtódnia a vágási pontjában specifikált minden egyes csatlakozási pontnál? Mi történik akkor, ha különböző aspektusok ugyanazt a vágási pontot definiálják? Ezek az elméletben és a gyakorlatban is problémát jelentenek? Az aspektusorientált tesztelést támogató olyan eszközökre van szükségünk, amelyek segítséget nyújtanak a tesztek lefedettségének mértékével kapcsolatos becslésekben.

Ahogy a 27. fejezetben említettem, a nagy projektek általában külön minőségi csapatokkal rendelkeznek. Ezek a csapatok fektetik le a minőségi szabványokat, és ők ellenőrzik, hogy a programok átvizsgálása és tesztelése a szabványoknak megfelelően történt. Az átvizsgálás és a tesztek származtatásának problémái jelentős mértékben akadályozzák az aspektusorientált szemléletmód alkalmazását a nagy szoftverfejlesztési projektekben.

Kulcsfogalmak

- A szoftverfejlesztés aspektusorientált megközelítésének egyik alapvető haszna az, hogy támogatja a vonatkozások szétválasztását. Ha a keresztező vonatkozásokat aspektusokként reprezentáljuk, lehetővé válik ezen vonatkozások egymástól független megértése, újrafelhasználása és módosítása.
- Keveredéssel akkor találkozunk, ha egy rendszer egy modulja különböző rendszerkövetelményekhez tartozó kódot tartalmaz. Hasonló jelenség a szétzóródás, ami akkor jelentkezik, mikor egyetlen vonatkozás implementációja több programkomponens között oszlik meg.
- Minden aspektus tartalmaz egy vágási pontot (egy olyan utasítást, amely definiálja, hogy az aspektust hol kell a rendszerbe szőni), és egy tanácsot, amely lényegében a keresztező vonatkozást implementáló kód. A csatlakozási pontok azok az események, amelyekre a vágási pontokból hivatkozni lehet.
- A vonatkozások szétválasztását támogatja a tervezésnek az a módja, amelyben a rendszert úgy tervezzük meg, hogy meghatározunk egy magot, amely a megrendelők elsődleges igényeit implementálja, a másodlagos vonatkozásokat pedig ezen mag kiterjesztéseiként valósítjuk meg.
- A vonatkozások azonosításához használhatunk nézőpont-orientált követelményfeltárást, amely során kiderítjük, mik a megrendelők követelményei, továbbá meghatározzuk a keresztező szolgáltatás minőségi és politikai vonatkozásokat.

- A követelményfeltárástól a tervezésre történő továbbhaladásához használati esetek azonosítása szükséges, ahol minden használati eset egy-egy megrendelői követelménynek felel meg. A terveket olyan modellekkel reprezentálhatjuk, amelyek az UML olyan kiterjesztett változatát használják, amelyben megtalálhatók az aspektusokat megjelenítő sztereotípiák.
- Az átvizsgálás és a tesztek származtatásának problémái jelentős mértékben akadályozzák az aspektusorientált szemléletmód alkalmazását a nagy szoftverfejlesztési projektekből.

További irodalom

„Aspect-oriented programming”. A CACM speciális kiadása számos cikket tartalmaz az általános hallgatóság számára, amelyek jó kiindulási pontként szolgálhatnak az aspektusorientált irodalomban való alaposabb elmélyedés előtt. (*Comm. ACM*, 44 (10), October 2001.)

Aspect-oriented Software Development. Többszerzős mű, számtalan cikket tartalmaz az aspektusorientált szoftverfejlesztéssel kapcsolatban, amelyeket a terület legelismertebb kutatói írtak. (Filman, R. E., Elrad, T., Clarke, S. és Askit, M., 2005, Addison Wesley.)

Aspect-oriented Software Development with Use-Cases. Gyakorlati könyv szoftvertervezők tollából. A szerzők kifejtik, hogyan használhatók a használati esetek a vonatkozások szétválasztására, továbbá ezek hogyan használhatók az aspektusorientált tervezés alapjaiként. (Jacobsen, I. és Ng, P., 2005, Addison Wesley.)

Feladatok

- 32.1. Milyen különböző típusú megrendelői vonatkozások merülhetnek fel nagy rendszerek fejlesztése során? Hogyan segíthetik az aspektusok az ilyen típusú vonatkozások implementációját?
- 32.2. Összegezze, mit jelent a keveredés és a szétszóródás. Példák használatával magyarázza meg, miért jelentenek a fenti jelenségek problémát a rendszerkövetelmények megváltozása esetén.
- 32.3. Mi a különbség a csatlakozási pont és a vágási pont között? Magyarázza meg, hogy ezek hogyan egyszerűsítik a kód beleszövését a programba, ezáltal a keresztező vonatkozások kezelését.
- 32.4. Mutassa meg, hogy a könyvben korábban leírt LIBSYS-rendszer hogyan tervezhető meg úgy, hogy a terv egy magrendszerből és a hozzá kapcsolódó kiterjesztésekből álljon. Milyen funkcionalitást tartalmaznának a kiterjesztések?
- 32.5. Milyen nézőpontokat kellene figyelembe vennünk egy, a 30. fejezetben megismert betegnyilvántartó rendszer követelményeinek meghatározása során? Mik lehetnek a legfontosabb keresztező vonatkozások?

- 32.6. A 32.9. ábra vázlatos funkcionalitás leírásának használatával határozzon meg hat további, a 32.11. ábrától különböző használati esetet a készlet-nyilvántartási rendszerhez. Ahol lehetséges, mutassa meg, hogyan szervezhetők ezek kiterjesztésekbe.
- 32.7. A 32.15. ábrán bemutatott aspektus sztereotípiá segítségével részletezze a 32.14. ábrán szereplő Rendelés és Monitor aspektusokat.
- 32.8. Magyarázza meg, hogyan léphet fel aspektusok között interferencia, és tegyen javaslatot arra, hogy ez elkerülhető legyen a tervezési folyamat során.
- 32.9. Magyarázza meg, hogy a vágási pont specifikációk mintákkal történő megadása miért nehezíti az aspektusorientált programok tesztelését és belövését. Ehhez gondolja végig, hogy a programtesztelésben milyen szerepet tölt be a program kimenetének összehasonlítása az elvárt kimenettel.
- 32.10. Magyarázza meg, hogyan egyszerűsítheti le az aspektusok használata a programok belövését.