Aspektus-orientált modellezést támogató UML-profil

Szerző: **Tasi Eszter**,
Babeş-Bolyai Tudományegyetem,
Matematika és Informatika Kar,
Informatika szak, IV. évfolyam

Témavezető : **Lect. Dr. Darvay Zsolt**,
Babeş-Bolyai Tudományegyetem,
Matematika és Informatika Kar,
Programozási Nyelvek és Módszerek Tanszék

XI. Erdélyi Tudományos Diákköri Konferencia - Kolozsvár,
2008. május 23-24.

1. Bevezetés

Egyre népszerűbb az aspektus-orientált programozás, azonban modellezéséhez nem alakult még ki szabvány, ezért az aspektus-orientált modellezést támogató eszközök száma nagyon kevés és ezek is hiányosak, a rendszereknek valamilyen sajátos szempontból való vizsgálatát teszik lehetővé. Munkám célja egy UML-profil készítése, mely lehetőséget nyújt az aspektus-orientál modellezésre a használati eset, osztály-, komponens-, szekvencia-, kommunikációs-, állapotgép- és aktivitásdiagramok segítségével.

A bevezetés további részében bemutatom az aspektus-orientált programozási paradigma alapvető elemeit, ezt követően az UML négyszintes architektúráját és bővítési mechanizmusait, majd felsorolom az aspektus-orientált szoftverfejlesztés előnyeit. A második rész az UML-profilt definiálja, bemutatja a bevezetett sztereotípusokat és megkötéseket, amit a profilnak egy gyakorlati alkalmazása követ.

1.1. Az aspektus-orientált programozás (AOP)

A számítógépes szoftverek az idők folyamán egyre komplexebbekké váltak, ezáltal karbantartásuk, átláthatóságuk egyre nehezebb feladatnak bizonyul. A jelenlegi rendszerek többségének implementációjára objektum-orientált programozási nyelveket használnak, ami lehetővé teszi a program alapfunkcióinak egységbe zárását. Azonban gyakori, hogy egy osztály, ezen belül egy metódus, nemcsak kimondottan a saját feladatát tölti be, hanem különböző mellékfunkciókkal bővül, mint például szálak szinkronizálása, naplózás, tranzakciók kezelése, különböző feltételek ellenőrzése, hibakezelés, melyek összekuszálják a kódot *(code tangling)*. Ezek a mellékfunkciók több osztály esetén is megjelennek, amik ugyanannak a kódrészletnek az ismétlését, valamint az azonos mellékfunkcióhoz tartozó kód *szétszóródását (code scattering)* eredményezik. A kapott kód, nehezen használható újra és nehezen módosítható, ha a mellékfunkciók valamelyikén szeretnénk változtatni.

A fenti problémák oka, az *átmetsző feladatok (crosscutting concerns)* jelenléte, azaz olyan feladatoké, amelyek nem tartoznak egyetlen alapfunkcióhoz sem, de néhányat érintenek belőle. A követelmények tere tehát n-dimenziós – minden átmetsző követelmény egy-egy dimenzió –, a jelenleg használt implementációs technikák azonban egydimenziósak. Így az n-dimenziót egyre kell leképezni, az alapfunkciók domináns dimenziójára, ami nem odatartozó kódrészek beékelését eredményezi.

Az AOP lehetővé teszi a több dimenziós követelményrendszer dimenziónkénti implementálását: az alapfunkciók implementálhatók a hagyományos módon, míg az átmetszők úgynevezett aspektusokként. A szétválasztott dimenziók kódjait az ún. Aspect Weaver *összeszövi (weave)* egy végleges végrehajtható programmá. A programozónak így csak az alapfunkciók implementálására kell összpontosítania, így munkájának minősége jobb lesz, teljesítménye megnövekszik. [1]

Az aspekts-orientált programozási nyelvek megvalósítása a már létező nyelvek aspektusokkal való kibővítését jelenti . A legelterjedtebb a XEROX Parc által fejlesztett Javan alapuló AspectJ. Az aspektus-orientált programozási nyelvek a következő új elemekkel bővültek [2, 3]:

Programpont (Join point). Egy program műveletek rendezett sora, amit változókon hajtunk végre. Hozzárendelhető egy gráf, ami a műveletek sorrendjét és az adatok közti összefüggést ábrázolja. Ezt nevezzük *adatfolyamgráfnak (Dataflow graph).* Általános értelemben ezen gráf csúcsait tekinthetjük programpontoknak, azonban ez programozási nyelvenként különbözhet. Például az AspectJ esetén programpont egy metódus, konstruktor meghívása vagy végrehajtása, egy mezőre való hivatkozás vagy annak beállítása, statikus inicializáló vagy objektum inicializáló ill. preinicializáló végrehajtása, kivételkezelő vagy tanács végrehajtása. Viszont nem számít programpontnak egy elágazási struktúrába vagy egy ciklusba való belépés.

Pontszűrő (Pointcut). A pontszűrő segítségével kiválasztható a programpontok egy halmaza. Az aspektus-orientált programozási nyelvek rendelkeznek *primitív pontszűrő kijelölőkkel*, melyek összetehetők logikai operátorokkal. Az összetett, komplexebb pontszűrők elláthatók névvel újrahasználás céljából. Szükség lehet a programpontoknál használt adatokra, információra, amit *kontextus*nak nevezünk. A pontszűrők segítségével a kontextus elérhetővé tehető a tanács számára.

Tanács (Advice). A tanács egy speciális metódus implementációja, mely végrehajtására több programpont esetén kerül sor. A kódrész például AspectJ esetén Javaban íródik. A kódon kívül szükséges megadnunk, hogy milyen pontszűrők esetén kell a kódot végrehajtani és ez a programpontok előtt, helyett, után vagy egy kivétel kidobásakor történik. A tanács átveheti és használhatja a kódban a pontszűrő által szolgáltatott

kontextust. Az összeszövés során a tanács kódja beillesztődik az összes olyan programpont elé/helyébe/mögé, amiket a pontszűrővel kiválasztottunk.

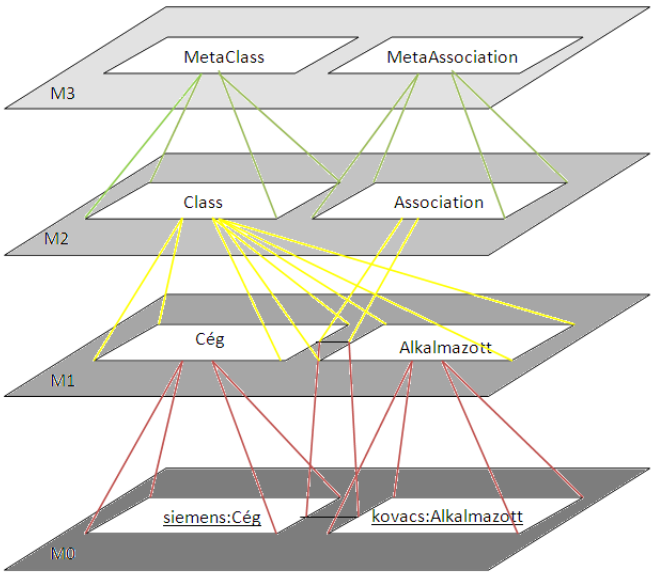
Aspektus (Aspect). Az aspektusok teszik lehetővé az átmetsző feladatok implementálását. Deklarálnunk kell őket, lehetnek adattagjai, metódusai, öröklődhetnek vagy implementálhatnak egy interfészt, akárcsak egy közönséges osztály, de ezen kívül tartalmazhat pontszűrőket és tanácsokat is. A pontszűrők segítségével definiáljuk, hogy *hol* metszi az aspektus síkja az alapfunkciók síkját, a tanácsok segítségével pedig azt, hogy *mivel* metszi. Az absztrakt aspektusok tartalmazhatnak absztrakt pontszűrőket, melyekről egy leszármazott aspektusban határozhatjuk meg, hogy milyen programpontokat szűrjenek ki. Alapbeállításként egy aspektusnak egyetlen példánya van, de ez deklaráláskor módosítható. Felállítható egy sorrend az aspektusok között, mely abban az esetben szükséges, ha egy programpont esetén a különböző tanácsok végrehajtási sorrendje nem tetszőleges.

Típusközi deklarációk (Inter-type declarations). Az eddigiekben az átmetsző feladatok dinamikus implementációját láttuk, ami pontszűrők és tanácsok segítségével történik. Azért dinamikus, mert az átmetszett komponens működésén végez módosításokat. Az AOP lehetőséget nyújt az osztálystruktúra és az osztályhierarchia módosítására is: egy aspektuson belül kibővíthetünk egy osztáyt új adattagokkal és metódusokkal, illetve egy osztálynak beállíthatunk új ősosztályokat vagy azt, hogy milyen interfészeket implementál és elláthatjuk őket a szükséges metódusok implementációjával.

1.2. Az UML bővítési mechanizmusai

Az UML (Unified Modelling Language) a szoftverfejlesztésben a rendszerek objektum-orientált modellezésére használt vizuális jelölési szabvány, amit az OMG (Object Management Group) specifikált.

Az UML-nek négy absztrakciós szintje van (1 ábra). Az M3 szinten található a meta-metamodell, ami a legmagasabb absztrakciós szintű elemeket – meta-metaobjektumokat – definiálja. A meta-metamodell egy nyelvet biztosít metamodellek specifikálására. Meta-metamodellre példa az OMG által definiált MOF (Meta-Object Facility). A MOF meta-metaobjektuma például a `MetaClass` és `MetaAssotiation`.



1. ábra. Négyszintes architektúra

Egy szinttel lennebb található a metamodell (M2), ami a meta-metamodell egy példánya. A metamodell modellek létrehozására szolgál. A metamodell tehát egy modellezési nyelv, ide tartozik az UML is, ami a MOF-nak egy példánya. A `Class`, `Assotiation` modellelemek a `MetaClass` és `MetaAssotiation` példányai. A MOF szintén UML jelölésekkel van definiálva, ezért az UML-t tulajdonképpen önmagával definiáljuk.

Az M1 szinten találhatók a szoftvertervező által létrehozott modellelemek, diagramok, melyek a metaobjektumok példányai. A legalsó szintet (M0) a definiált osztályok példányai, azaz az objektumok – a valós világ elemei – alkotják. [5]

Az UML bővítésére két lehetőség van. Módosíthatjuk az UML metamodelljét a MOF segítségével, ezt *nehézsúlyú (heavyweight)* bővítésnek nevezzük. Hátránya, hogy nem kompatibilis a már létező UML-en alapuló modellezési szoftverekkel. Másik lehetőség a *könnyűsúlyú (lightweight)* bővítés az UML három bővítési mechanizmusának – a *sztereotípusok*, *tagged value*-k és *megszorítások* – használatával.

A sztereotípus egyfajta metaosztály, amely az UML egy vagy több metaosztályát bővíti, azzal a megkötéssel, hogy nem használható külön a kibővített metaosztály valamelyike nélkül. Magába foglalja a kibővített metaosztály tulajdonságait, amelyek új tulajdonságokkal – metaattributumokkal vagy *tagged value*-kkal – bővíthetők. A sztereotípusokra megkötések adhatók az OMG által specifikált OCL (Object Constraint Language) (specifikáció a [6]-ban) segítségével.

Egy profil egy speciális csomag, mely doménspecifikus sztereotípusokat, azoknak új tulajdonságait és megszorításokat tartalmaz. Egy profil kompatibilis az UML metamodellel, mert nem módosítja a már létező megkötéseket és asszociációkat a metaosztályok között, nem hoz létre új metaosztályokat, így könnyen használható a már létező CASE-eszközökkel, könnyen exportálhatók az XMI segítségével. Ahhoz, hogy egy modell, amit egy adott profil felhasználásával definiáltunk, *helyes* legyen, eleget kell tegyen a megadott megszorításoknak.

1.3. Aspektus-orientált modellezés (AOM)

Az aspektus-orientált megközelítés használata a programozás szintjén túlmenően, a szoftverfejlesztés összes fázisában előnyös. Az átmetsző feladatok korai azonosítása

- *növeli a rugalmasságot*: a szoftver egyes követelményei könnyebben módosíthatók lesznek, mivel ezek külön egységbe vannak zárva;
- *egyszerűbbé teszi új funkciók bevezetését*, mert ez egy új külön álló egység létrehozását jelenti;
- *javítja a minőséget*: lehetőség nyílik arra, hogy a fejlesztő a rendszernek egyszerre csak egy követelményével foglalkozzon, elemezze, tervezze;
- *csökkenti a fejlesztés időtartamát*, mivel a különböző követelményeknek jól elkülöníthető, lazán összekapcsolt tervezési egységek felelnek meg, melyeket különböző csapatok fejleszthetnek egyidejűleg, egymás közötti minimális kapcsolatfenttartással;
- *növeli az újrahasznosítás lehetőségét*: a teljesen különálló egységek, könnyebben felhasználhatók más projektekben is.[8]

A szoftverfejlesztés során fontos szerepet játszik a rendszernek különböző szempontokból való vizsgálata, modellezése. Született néhány kísérlet aspektus-orientált modellezést támogató eszközök készítésére, azonban még nem alakult ki egy szabvány.

Az esetek többségében a kutatók egyetértenek abban, hogy mivel az UML az OOP általánosan elfogadott modellezési nyelve és az AOP az OOP-n alapszik, ezért az UML-t kell új metamodell-elemekkel és jelölésekkel kibővíteni, olyan módon, hogy az AOM-re is

alkalmassá váljon. Az ajánlatok jelentős része UML-profil definiálását jelenti, más kutatók inkább a nehézsúlyú bővítés mellett szavaznak. Van példa teljesen új modellezési nyelv meghatározására is a MOF segítségével.

A kutatók más-más célokkal definiáltak aspektus-orientált modellezési nyelveket: egyesek a rendszer-architektúra modellezésére fektették a hangsúlyt [10, 11, 12], míg más esetben a működés vagy az állapotok modellezése volt a fontosabb [13]. Született olyan elképzelés, mely kimondottan az AspectJ elemeivel bővítette az UML-t [11], más esetben egy nagyon általános modellezési nyelv készítése volt a cél, ami az AOP-vel rokon programozási paradigmákkal (szubjektiv, adaptív programozás, hiperterek) is kompatibilis [10]. Ezek a megközelítések külön-külön túlságosan specifikusak (vagy túlságosan általánosak), azonban figyelembe véve őket felépíthető egy aspektus-orientált modellezési nyelv, mely alkalmas általános használatra. Ez az általam készített UML-profil *célja*.

2. Aspektus-orientált modellezést támogató UML-profil

Az UML-profilom lehetővé teszi az aspektus-orientált modellezést használati eset (use case), osztály-, komponens-, szekvencia-, kommunikációs-, állapotgép- és aktivitásdiagramok esetén. Nem kimondottan az AspectJ-be bevezetett új programelemek alapján építettem fel a modellezési nyelvet, mert ez túlságosan specifikus. Viszont tény, hogy az AspectJ-jellegű nyelvek nagyobb teret hódítottak, mint az adaptív programozás vagy a hiperterek, tehát nem érdemes annyira általános modellezési nyelvet készíteni, ami ezekkel a megközelítésekkel is összefér.

A profil készítéséhez a Borland Together Architect 2006 for Eclipse CASE-eszközt használtam. A szoftver tulajdonképpen az Eclipse keretrendszert kibővítő pluginek halmaza és az elkészített profil is pluginként telepíthető. A Borland Together szab néhány olyan korlátot a profildefiniálás esetén, melyek nem szerepelnek az UML-specifikációban:

- absztrakt metaosztályok nem bővíthetők sztereotípusokkal,
- ha egy sztereotípus két metaosztályt bővít, nem jeleníthető meg,
- megszorításokat csak a metaosztályok kontextusában lehet írni, mely következtében a megkötések átláthatósága és megírása nehezebb feladattá válik.

2.1. AO-elemek használati eset (use case) diagramok esetén

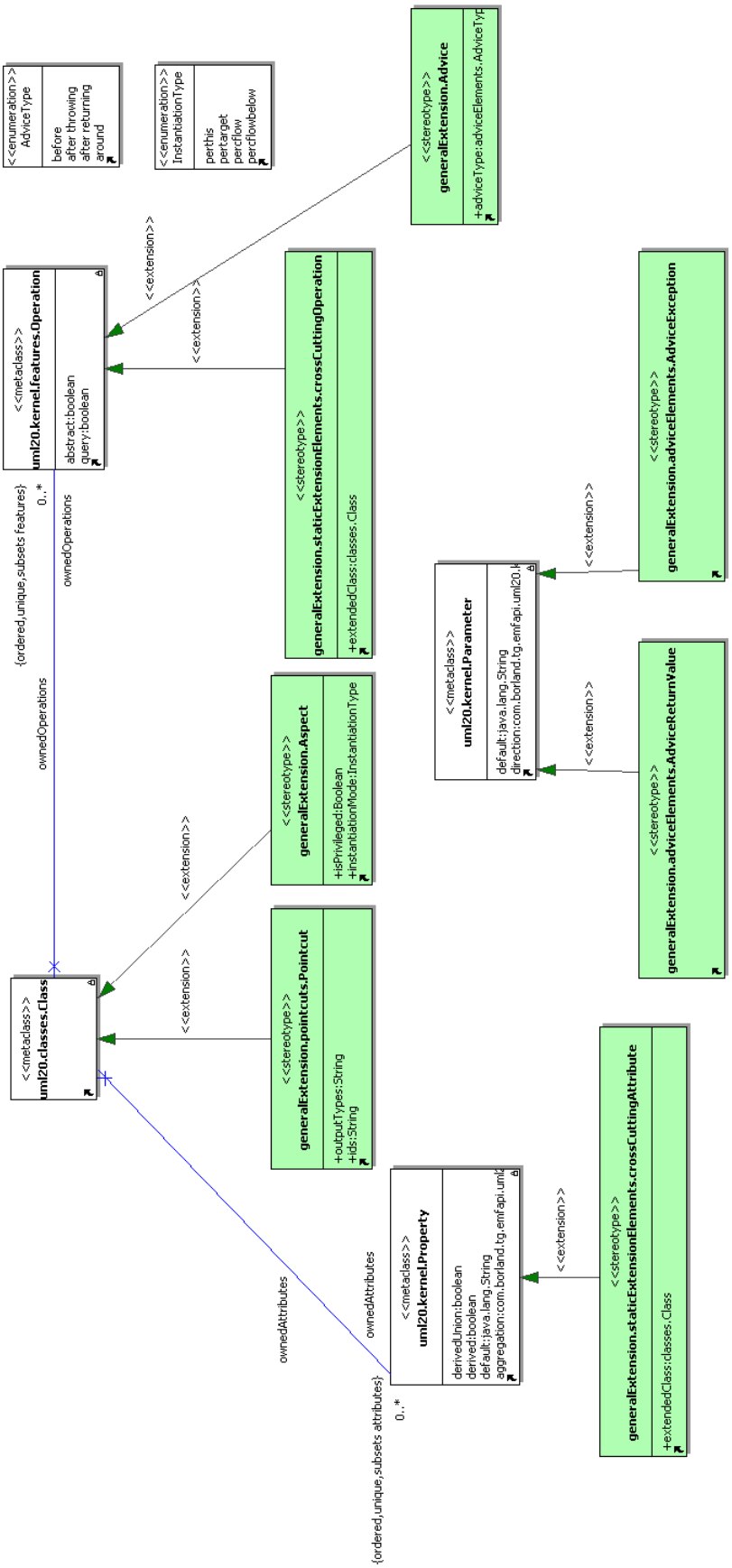
Ivar Jacobson – a használati eset diagramok megteremtője – már régen felismerte a mellékfeladatok beékelődésének problémáját, ezért vezette be a *kiterjesztés (extension)* és a *kiterjesztési pont (extension point)* fogalmát. Az előbbi egy kapcsolat két használati eset között a következő jelentéssel: a forrás használati eset kibővíti a cél használati eset viselkedését. Ez a kibővítés kiterjesztési pontokban lehetséges, melyek a cél használati eset viselkedésében adott pontok. A használati eset diagram elemei megfeleltethetők az AO[1] elemeivel: egy használati eset egy feladat; azok a használati esetek, melyek több másikat terjesztenek ki, átmetsző feladatok; a kiterjesztett használati esetek az alapfeladatok (ha nem terjesztenek ki más használati eseteket). A kiterjesztési pontok a programpontok egy elvontabb vátozatai. [7]

A használati eset diagramok esetén egyetlen sztereotípust vezettem be: `CrossCuttingConcern`, mely az `uml20.usecases.UseCase`[2] metaosztályt bővíti és az átmetsző feladatok modellezését szolgálja, így jobban kihangsúlyozható, hogy mely használati esetek átmetsző feladatok és melyek alapfeladatok. Megszorítás: a nem átmetsző használati esetek, nem terjeszthetnek ki más használati eseteket[3].

2.2. AO-elemek az osztálydiagramok esetén

2.2.1. AO-elemek csomagok szintjén

Az `uml.kernel.packages.Package` metaosztályt bővíti a `CrossCuttingPackage` sztereotípus, mely muszáj aspektust tartalmazzon, mivel egy átmetsző feladatot megvalósító modellelemek halmaza. Ha egy csomag nem átmetsző, akkor nem tartalmazhat aspektusokat.[4]



2. ábra. Alapvető aspektus-orientált elemek

2.2.2. Alapvető AO-elemek (2 ábra)

Az *aspektusok* modellezését az `Aspect` sztereotípussal teszem lehetővé, mely az `uml20.classes.Class` metaosztályt bővíti. A sztereotípusnak van két új tulajdonsága, az egyik az `isPrivileged`, ami ha igazra van beállítva az aspektus a típusközi deklarációk során hozzáfér a bővített osztályok privát elemeihez is. Az `instantiation` attributummal az aspektus példányosítási módja adható meg. Ha nem állítjuk be az aspektusból egy példány jön létre, különben az `InstantiationType` felsorolás valamelyik literálja választható ki értéknek. Egy aspektusnak kötelező átmetsző elemet tartalmaznia, mely egy `Advice`-t (lásd alább), `crossCuttingAttribute` sztereotípusú mezőt vagy `crossCuttingOperation` sztereotípusú metódust (lásd 2.2.4) jelent.

A *tanács* egy speciális operáció, tehát az `Advice` az `uml20.kernel.features.Operation` sztereotípusa. Muszáj megadni a típusát (`adviceType`), mely az `AdviceType` felsorolás valamelyike. Csakis aspektusok tartalmazhatják, nincs elő- és utófeltétele, nem öröklődhet és nem lehet absztrakt. A tanács hagyományos paraméterei (a kontextus) bemeneti paraméterek. Deklarálhatunk speciális paramétert is, mely sztereotípusa `AdviceReturnValue` vagy `AdviceException`, melyek az `uml20.kernel.Parameter` metaosztály sztereotípusai. Az `AdviceReturnValue` ki- és bemeneti paraméter, az `after returning` típusú tanács deklarálhat ilyen paramétert, mely által hozzáférhet a végrehajtott programpont kimeneti értékeihez. Az `AdviceException` bemeneti paraméter, az `after throwing` típusú tanács esetén kötelező a jelenléte, azt határozza meg, hogy milyen kivétel(ek) kidobásakor kell a tanácsnak végrehajtódnia.

2.2.3. Pontszűrők (3 ábra)

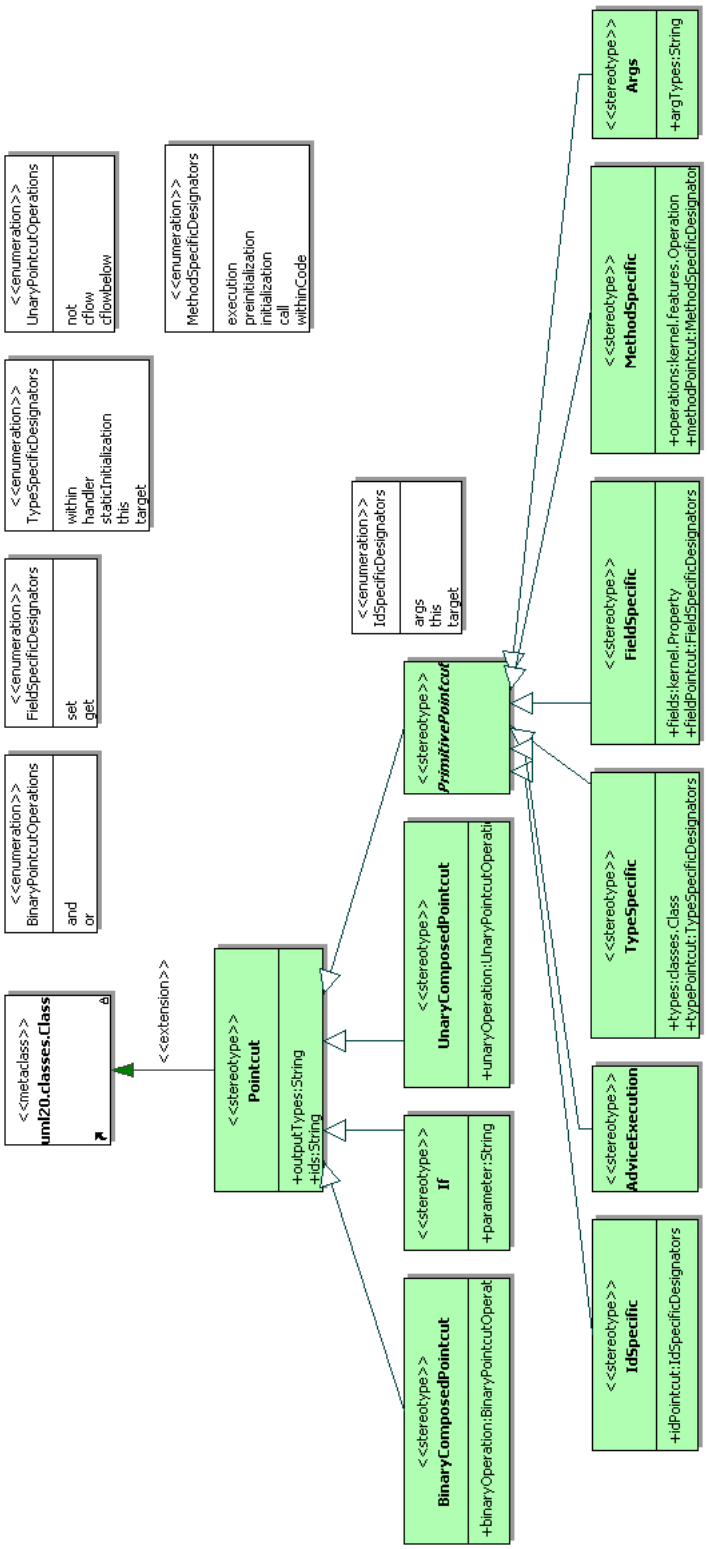
A profil a pontszűrők szintjén az AspectJ programozási nyelv által szolgáltatott lehetőségeket nyújtja, leszámítva a wildcard karakterek használatát, mely modellezés szinten nem elég kifejező. Az AspectJ primitív pontszűrő kijelölőit csoportosítottam a követ-

[1] A következőkben az AO rövidítés az aspektus-orientált kifejezésre

[2] A Borland Together metamodelljének metaosztályairól van szó, mely helyenként különbözik a standard UML metamodelltől. Az uml csomag tartalmaz néháy alapvető metaosztályt, mint például az Element vagy NamedElement metaosztályok, míg az uml20-ban található az UML 2.0 verzió modellelemeinek többsége.

[3] A megkötések a profil esetén OCL-ben íródtak, a dolgozatban a könnyebb megértés érdekében természetes nyelvet használtam.

[4] A Borland Together az osztálydiagramokon belül nyújt lehetőséget a csomagok modellezésére is.



3. ábra. Pontszűrők

Kategória	Sztereotípus	Típus-jelölő metaattributum	Felsorolás	Paraméter-jelölő metaattributum	Paramétertípus
típus-specifikus	TypeSpecific	typePointcut	TypeSpecificDesignators	types	Class
metódus-specifikus	MethodSpecific	methodPointcut	MethodSpecificDesignators	operations	Operation
mező-specifikus	FieldSpecific	fieldPointcut	FieldSpecificDesignators	fields	Property
id-specifikus	IdSpecific	idPointcut	IdSpecificDesignators	ids, outputTypes	String

1. táblázat. A pontszűrő-kategóriák és sztereotípusok közötti megfeleltetés

kező kategoriákba az alapján, hogy milyen adattípust kell paraméternek megadnunk:

- *típus-specifikus pontszűrők*, paraméterei osztályok, ide sorolhatók a `within`, `handler`, `staticInitialization`, `this`, `target` pontszűrők;
- *metódus-specifikus pontszűrők*, paraméterei metódusok és konstruktorok, ebbe a kategóriába tartoznak az `execution`, `call`, `withincode`, `preinitialization` és `initialization` pontszűrők;
- *mező-specifikus pontszűrők*, paraméterei mezők, a `get` és `set` pontszűrők alkotják ezt a csoportot;
- *id-specifikus pontszűrők*, paraméterei változók és annak típusai és a kontextust szolgáltatják, ide sorolhatók a `this`, `target`, `args` pontszűrők.

Ezen kívül az `adviceexecution` pontszűrőnek nincs paramétere, a `cflow` és `cflowbelow` pontszűrők bemenete egy másik pontszűrő, ezért ezeket unáris operátoroknak tekintem a `not` mellett. Az `if` bemenete egy speciális logikai kifejezés. Bináris operátorok az `and` és az `or`.

A `Pointcut` sztereotípus esetén szintén az `uml20.classes.Class` metaosztályt választottam bővített metaosztálynak. Nem példányosítható, nincs leszármazottja, nincsenek metódusai és mezői, nem vehet részt asszociációkban. A `Pointcut` osztályon, interfészen vagy aspektuson belül létezhet. Ha nem absztrakt egyetlen alosztályt kell tartalmaznia, mely szintén pontszűrő, egyébként nincs alosztálya.

A `Pointcut` leszármazottjai az absztrakt `PrimitivePointcut`, `UnaryComposedPointcut`, `BinaryComposedPointcut` és az `If` sztereotípusok. Mindenik kategóriának és az `adviceexecution` pointcutnak a `PrimitivePointcut` egy-egy leszármazottja felel meg, minden kategória esetén szükséges kiválasztanunk, hogy pontosan melyik pontszűrőt szeretnénk használni a megfelelő felsorolás elemei közül, és meg kell adnunk a paramétereket

(lásd 1 táblázat). Az `args` sztereotípus segítségével olyan pontszűrő modellezhető, mely az adott típusú paramétereket használó programpontokat szűri ki. A primitív pontszűrők nem tartalmazhatnak alosztályokat, tulajdonosuk egy `Pointcut`, `BinaryComposedPointcut` vagy `UnaryComposedPointcut` lehet.

A `UnaryComposedPointcut` illetve `BinaryComposedPointcut` esetén szükséges megadnunk egy `UnaryPointcutOperations` típusú operátort illetve egy `BinaryPointcutOperations` típusú logikai operátort. Ezek a sztereotípusok tulajdonosa is a `Pointcut`, `BinaryComposedPointcut` vagy `UnaryComposedPointcut` valamelyike. Az `If` sztereotípus esetén a logikai kifejezés sajnos csak stringként adható meg.

Egy definiált `Pointcut` újrahasználható `UnaryComposedPointcut` illetve `BinaryComposedPointcut` esetén a `uses` sztereotípusú függőség segítségével, mely a `uml20.classes.Dependency` sztereotípusa. A `UnaryComposedPointcut` egyetlen alosztályt vagy `uses` függőséget tartalmazhat, míg a `BinaryComposedPointcut` esetén az alosztályok és `uses` függőségek számának összege 2. A `uses` sztereotípus esetén a forrás mindig egy `UnaryComposedPointcut` vagy `BinaryComposedPointcut`, a cél pedig egy `Pointcut` sztereotípus.

A `Pointcut` két metaattributuma az `ids` és az `outputType`, melyet örökölnek a leszármazottjai is, a kontextust szolgáltatják, az egyik a változók nevének sora, a másik ezeknek a típusa. A Borland Together korlátai miatt sajnos ez csak két rendezett stringsorral valósítható meg. Kontextust csak az `IdSpecific` sztereotípusú elemek szolgáltathatnak, ezért más primitív pontszűrő esetén ezt a két metaattributumot nem kell definiálni. A `UnaryComposedPointcut` esetén a metaattributum értékei azonosak kell legyenek az alpontszűrő metaattributumaival, ha az operátor nem a `not`. Az utóbbi esetben a sztereotípus e két attributuma határozatlan lesz az alpontszűrőtől függetlenül. `BinaryComposedPointcut` esetén `or` operátort használva a sztereotípus kontextusparaméterei a két alpontszűrő paraméterhalmazának metszete lesz, míg `and` esetén ezeknek egyesítése.

A pontszűrők megközelítése a `Class` metaosztály sztererotípusaként erőltetettnek tűnhet, de a lehetőségek felmérése után, ezt találtam a legmegfelelőbbnek. Az UML-specifikációra alapozva az `Expression` metaosztály tűnt a pontszűrő ábrázolására a legkézenfekvőbbnek, hisz ez a `ValueSpecification` metaosztállyal együtt egy operátorokból és operációkból álló kifejezésfát valósít meg. Azonban a Borland Together metamodellje nem tartalmazza ezeket a metaosztályokat. Másik lehetőség a `Feature` metaosztály

bővítése, hisz a pontszűrő az osztályok, aspektusok része, azonban ez egy absztrakt metaosztály, amit nem bővíthet sztereotípus. Másik lehetőség a `Feature` leszármazottjának, az `Operation`-nak használata, ami viszont egy osztálynak a működését, viselkedését valósítja meg, a pontszűrőnek azonban nem ez a célja.

A pontszűrők tulajdonképpen osztályoktól és aspektusoktól függetlenül létezhetnének, magasabb rendű modellelemek, melyek a többi osztályokkal, objektumokkal, metódusokkal operálnak. A programozási nyelvekben viszont csak aspektusokon és osztályokon belül létezhetnek, és ha másképp modelleznénk őket, túlságosan nagy lenne a szakadás a modell és implementáció között.

2.2.4. Statikus átmetszés modellezése (2, 4 ábrák)

A `crossCuttingOperation` sztereotípusú operációk valamint `crossCuttingAttribute` sztereotípusú attributumok csak aspektusok által deklarálhatók és esetükben kötelező megadni, hogy milyen osztályt bővítenek (`extendedClass`). Az attributumok neve és az operációk fejléce nem egyezhet a bővített osztályban deklarált mezők illetve operációk neveivel. A `crossCuttingOperation` és a `crossCuttingAttribute` a típusközi metódus- illetve meződeklarációt teszik lehetővé.

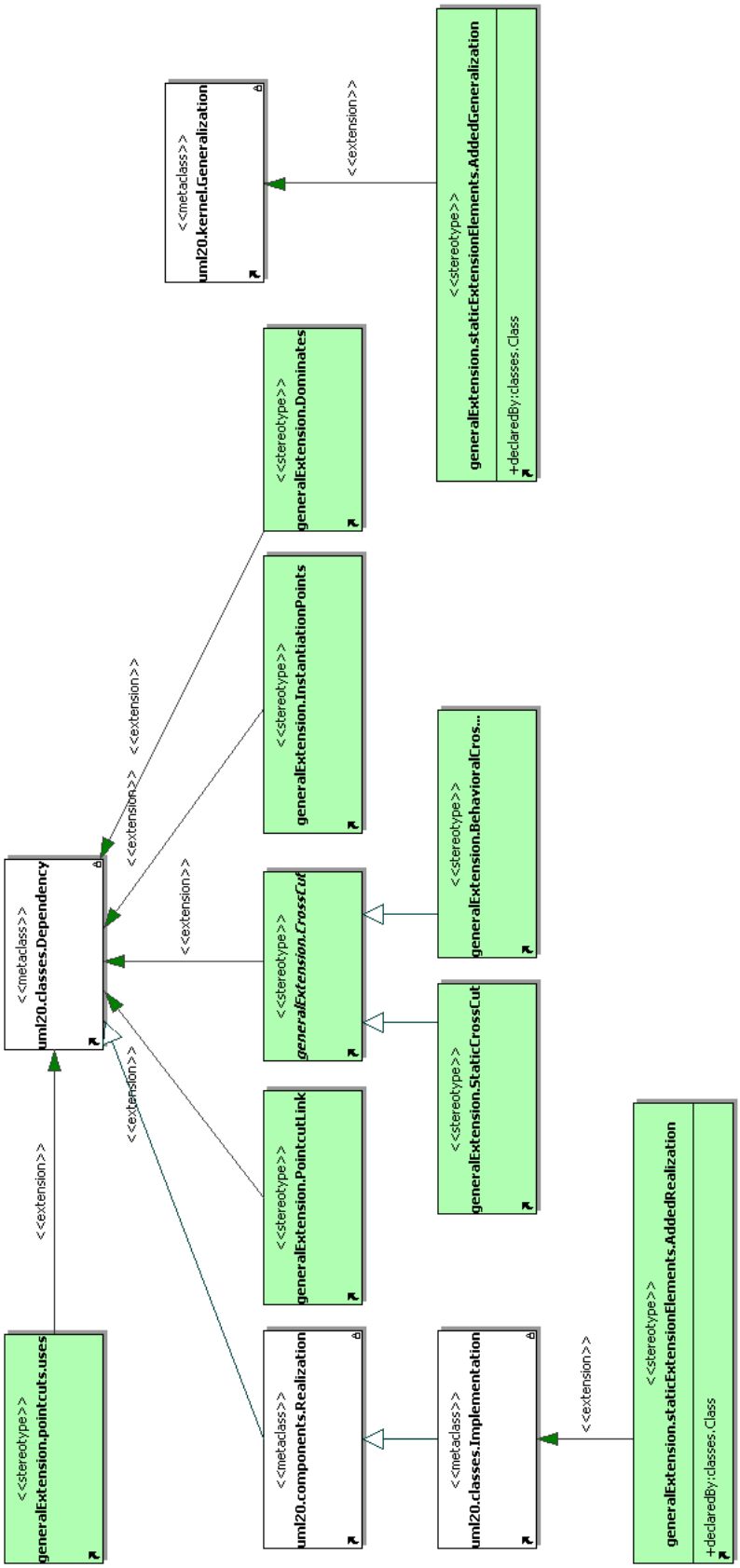
Az `AddedGeneralization` az `uml20.kernel.Generalization` metaosztályt bővítő sztereotípus, tehát öröklődést jelent, melyet a `declaredBy` metaattributum által meghatározott aspektus(ok) deklarál(nak). Hasonlóképpen az `AddedRealization` segítségével, mely az `uml20.classes.Implementation` sztereotípusa, implementálási relációt lehet meghatározni, melyet szintén egy vagy több aspektus deklarálhat.

2.2.5. Kapcsolatok (4 ábra)

Az aspektus-orientált modellezést támogató kapcsolatok az `uml20.classes.Dependency` metaosztály sztereotípusai, mert különböző típusú függőségeket fejeznek ki.

A `Dominates` sztereotípus két aspektus között létezhet: a forrás aspektus tanácsai a cél aspektus tanácsai előtt hajtódnak végre. Az aspektusokból és `Dominates` kapcsolatokból alkotott irányított gráfnak aciklikusnak kell lennie.

Az `InstantiationPoints` forrása egy `Aspect`, célja egy `Pointcut` sztereotípus. Ha egy aspektus példányosítási módja nem az alapértelmezett (az `instantiationMode` mező nem üres), akkor az aspektus kötelező módon kell tartalmazzon egy `InstantiationPoints`



4. ábra. Kapcsolatok

függőséget, mert a példányosításhoz szükséges megadni egy pontszűrőt.

A `PointcutLink` egy `Advice` és egy `Pointcut` között létezhet, megjelölve egy adott tanács esetén, hogy melyik pontszűrőt használja. Egy tanács pontosan egy `PointcutLink` függőségnek kell a forrása legyen. A forrás tanácsnak a kért kontextusa, meg kell egyezzen a cél pontszűrő által nyújtott kontextussal, azaz a tanács nem speciális paramétereinek neve és típusa azonos kell legyen a `Pointcut` `ids` és `outputTypes` attributumokban nyújtott paraméterek nevével és típusával.

A `BehavioralCrossCut` és a `StaticCrossCut` a `CrossCut` absztrakt sztereotípus leszármazottjai. Az első sztereotípussal rendelkező függőség forrása egy tanács, pontszűrő vagy aspektus, célja egy aspektus vagy egy osztály és – ahogy a neve is mutatja – viselkedésbeli átmetszést jelent. A `StaticCrossCut` egy aspektus és egy osztály, interfész vagy aspektus között létezhet, de csak abban az esetben, ha az aspektusnak van legalább egy `crossCuttingAttribute` vagy `crossCuttingOperation` sztereotípusú metódusa vagy mezője, amely esetén az `extendedClass` a célosztályt jelöli.

2.3. AO-elemek interakciós diagramok esetén (5 ábra)

Az interakciós diagramokat, azaz a szekvencia- és a kommunikációs diagramokat, megfelelővé kell tenni az aspektusok és obejktumok interakciójának szemléltetésére.

Ahhoz, hogy egy közönséges objektum életvonala megkülönböztethető legyen az aspektusétól, létrehoztam az `uml20.interaction.Lifeline` `AspectLifeline` sztereotípusát.

A kommunikációt az objektumok között az `uml20.interaction.Int_Communication-Link` metaosztály jelöli. Ezek a kapcsolatok megfeleltethetőek egy-egy programpontnak. Azért, hogy megadható legyen, hogy pontosan milyen programpontról van szó, a `uml20.interaction.Int_CommunicationLink` metaosztályt új sztereotípusokkal láttam el:

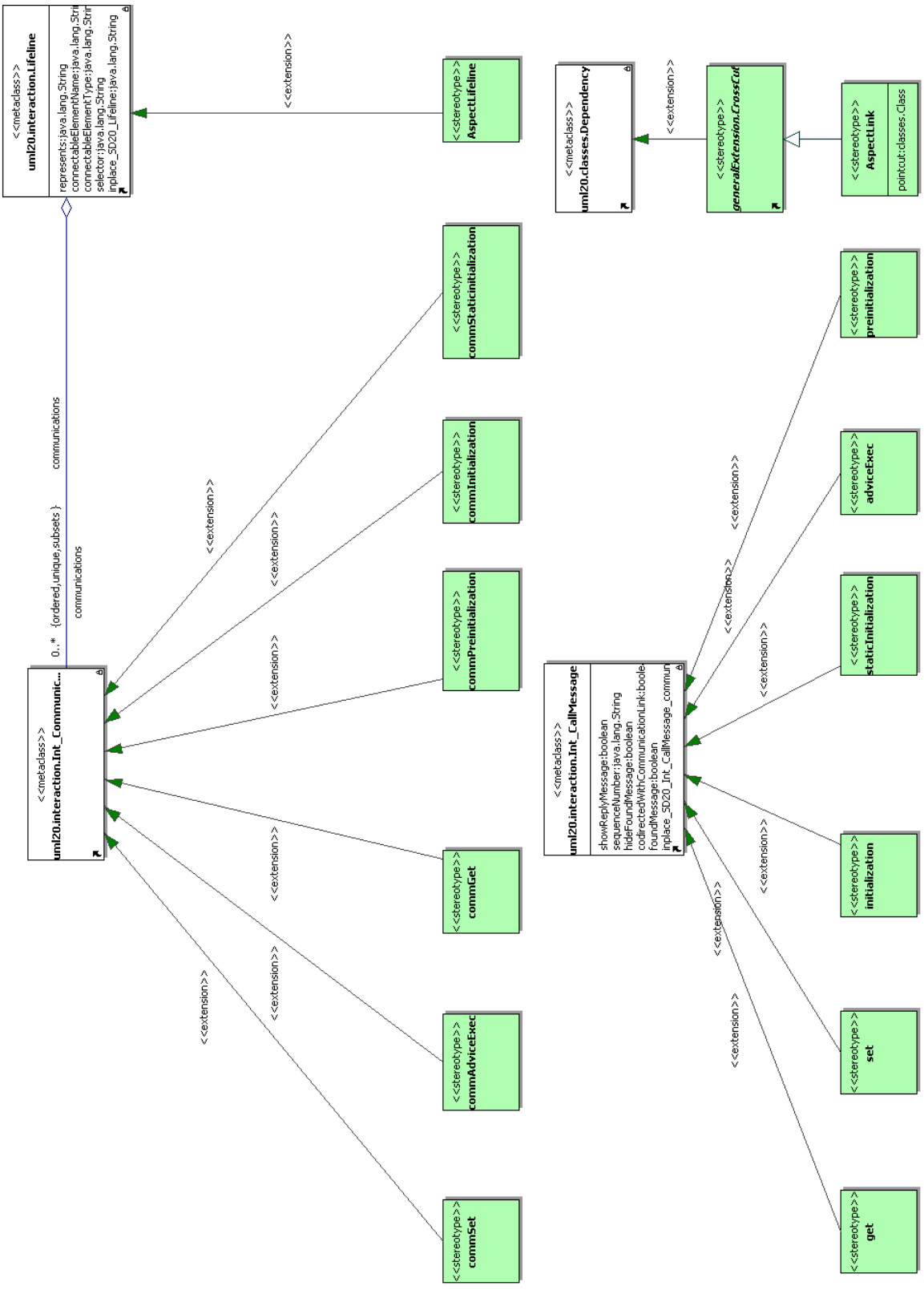
a. `commSet`, a mezőérték-állítás,

b. `commGet`, a mezőérték-lekérdezés,

c. `commAdviceExec`, a tanácsvégrehajtás,

d. `commPreinitialization`, az előinicializálás,

e. `commInitialization`, az inicializálás,



5. ábra. Interakciós AO-elemek

f. `commStaticInitialization`, a statikus inicilizálás programpontoknak felel meg.

Ha a kommunikációs kapcsolat sztereotípusa egyik sem az előzőek közül, akkor egy metódushívást jelent. A d, e, f sztereotípusok esetén a kapcsolat forrása és célja azonos. A `commAdviceExec` kapcsolat célja egy aspektus-életvonal. A kommunikációs diagramon ezek a kapcsolatok jeleníthetők meg az objektumok és aspektusok között.

A szekvenciadiagram esetén az objektum-életvonalak között bár létezik kommunikációs kapcsolat, ezek csak hívásüzenetek segítségével jeleníthetők meg, mely metaosztálya: `uml20.interaction.Int_CallMessage`. Ezért szükséges volt a fentieknek megfelelő sztereotípusok bevezetésére az `Int_CallMessage` esetén is (`get`, `set`, `initialization`, `preinitialization`, `staticInitialization`, `adviceExec`). Egy üzenet forrása az objektum-életvonalon található `Int_InvocationSpecification` modellelem, célja a célobjektum-életvonalon található `ExecutionSpecification`.

Hasznosnak találtam az aspektusok más stílusú megjelenítését is a szekvenciadiagramokon. Ebben az esetben nem látható az aspektus életvonala, a hangsúly azon van, hogy milyen pontokban metszi egy aspektus az egy alapfeladatot ellátó objektumok interakciójának síkját. Ehhez szükség volt a `Dependency` metaosztály esetén egy újabb sztereotípusra, az `AspectLink`-re. Az `AspectLink` egy `Aspect` sztereotípus és egy `Int_InvocationSpecification` vagy egy `ExecutionSpecification` metaosztály között létezhet, annak függvényében, hogy az aspektus egy `call` vagy egy `execution` pontszűrőt használ. Az `AspectLink` esetén a `pointcut` metattributum segítségével beállítható, az aspektus által használt `Pointcut`, mely az illető programpontot kiszűri.

2.4. AO-elemek más diagramok esetén

Az *aktivitásdiagramok* esetén a hangsúly azon van, hogy hogyan, milyen műveletekkel valósítható meg egy viselkedés, egy használati eset, elhanyagolva a műveleteket végző objektumokat. Tehát ebben az esetben értelmetlen az aspektusok vagy pontszűrők fogalmának bevezetése, azonban lehetőség nyílik az átmetsző műveletek azonosítására (azok a műveletek, melyek több tevékenység esetén megjelennek vagy egy tevékenységen belül többször szerepelnek és mégsem zárhatók külön tevékenységbe). Ezért definiáltam a `crossCuttingAction` sztereotípust, mely az `uml20.activities.Action` metaosztályt bővíti.

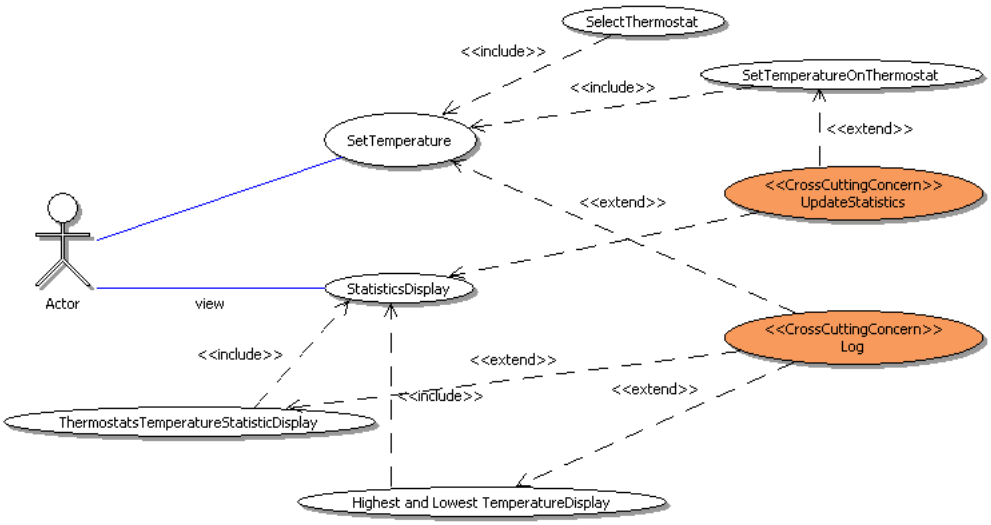
Az *állapotgépdiagramok* egy objektum belső állapotainak és az ezek közötti átmenetet kiváltó események, metódusok szemléltetését szolgálják. Az állapotgépként működő objektumok esetén a kiválasztott programpontok kifejezőbben ábrázolhatók állapotgépdiagramon, mint osztály- vagy szekvenciadiagramon. Ezért az állapotdiagramok esetén is lehetővé tettem az aspektusok használatát, és bevezettem a `StateMachineCrosscut` függőség-sztereotípust, mely forrása egy aspektus, célja egy állapot – `uml20.statemachines.State` – vagy állapotok közti átmenet – `uml20.statemachines.TranzitionLink`. A `StateMachineCrosscut` esetén is megadható, hogy melyik pontszűrő feladata az adott programpont kiválasztása.

A *komponensdiagramok* a komponensek közötti kapcsolatokat szemléltetik. A komponensek portokon keresztül kérnek és nyújtanak interfészeket és akkor működhetnek együtt, ha a nyújtott interfészek kielégítik a kért interfészeket. Az aspektusok bevezetése változtathat a helyzeten, ugyanis az aspektusok deklarálhatnak egy osztályt adott interfészt implementáló osztályként és implementálhatják a metódusokat. Tehát egy aspektus segítségével létesíthető kapcsolat olyan komponensek között is, melyek nem nyújtanak megfelelő interfészeket! Így az aspektus-orientált modellezés esetén nemcsak az interfészek lehetnek összekötő láncszemek, hanem az aspektusok is, ezért fontos az aspektusok megjelenítése a komponensdiagramok esetén.

A kutatók nagy része a rendszerek szerkezeti aspektus-orientált modellezésével foglalkozik, kevesebben a szekvencia- és az állapotgépdiagramok szintjén is bevezették az AO-elemeket, azonban a többi UML-diagram elkerülte figyelmüket.

3. Alkalmazás

A következő specifikációval rendelkező rendszer modellezésével bemutatom, hogyan alkalmazhatók a profil által bevezetett modellelemek: *Adott egy fűtőrendszer, mely több hőszabályzóval rendelkezik, amik segítségével különböző szobákban lehet állítani a hőmérsékletet. Szeretnénk egy olyan szoftvert készíteni, mely segítségével állíthatók a hőszabályzók. A hőmérséklet a következő kategóriák valamelyikébe sorolható: alacsony, közpes, magas és nagyon magas. A szoftver ki kell jelezze, hogy melyik kategóriába hány hőszabályzó hőmérséklete esik, valamint lehetőség kell legyen a legnagyobb és legkisebb hőmérséklet megtekintésére. Szeretnénk lementeni a hőszabályzó-változtatásokat és az ezt eredményező*



6. ábra. Use Case diagram. A hőszabályzó beállítása a megfelelő hőszabályzó kiválasztásából és annak állításából áll. Az állítás következtében a statisztikakijelzőt is kell módosítani. Az UpdateStatistics tehát kiterjeszti a hőmérsékletállítás használati esetet, közben a statisztikakijelzőt használva. A StatisticsDisplay szintén két al használati esetből áll. A Log használati eset átmetszi ezeket és a hőmérsékletállítás használati esetet is.

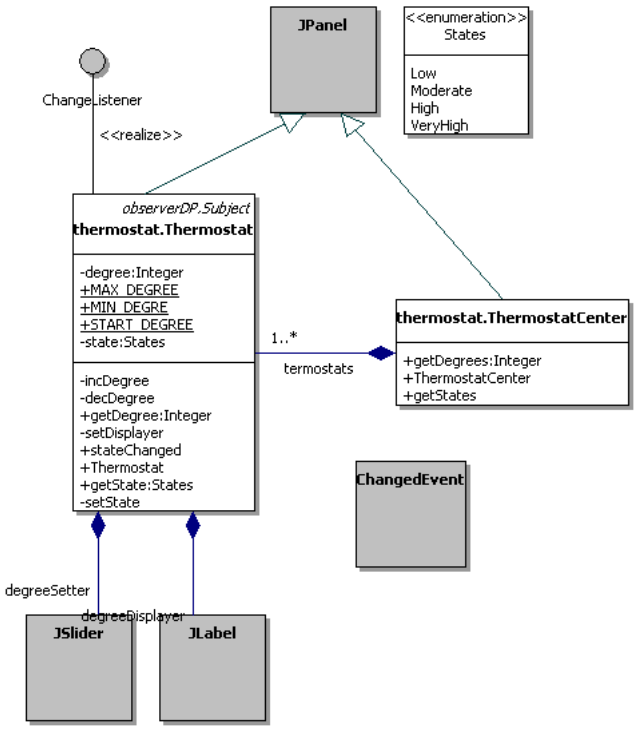
statisztika-módosulásokat.

Azzal, hogy fizikailag hogyan valósul meg a hőszabályzók állítása, nem foglalkozom.

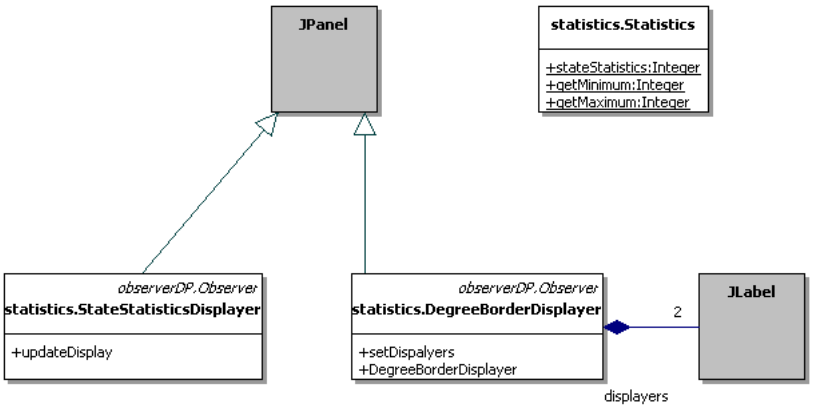
A használati eset diagramon (6 ábra) láthatók az asonosított használati esetek. Megfigyelhető, hogy az `UpdateStatistics` és a `Log` használati esetek kiterjesztenek más használati eseteket, ezért ezeket a `CrossCuttingConcern` sztereotípussal láttam el.

A statisztikák automatikus frissítésének megvalósítára a Megfigyelő tervezési minta használata előnyös. Az `observerDP` csomag a tervezési minta általános megvalósítása, a `thermostat` a hőszabályzók kezelését, míg a `statistics` csomag az adatok feldolgozását és megjelenítését biztosítja. Az utóbbi két csomag osztálydiagramjára nem térek ki (lásd 7 és 8 ábrák), ezek a hagyományos objektum-orientált modellezést használják.

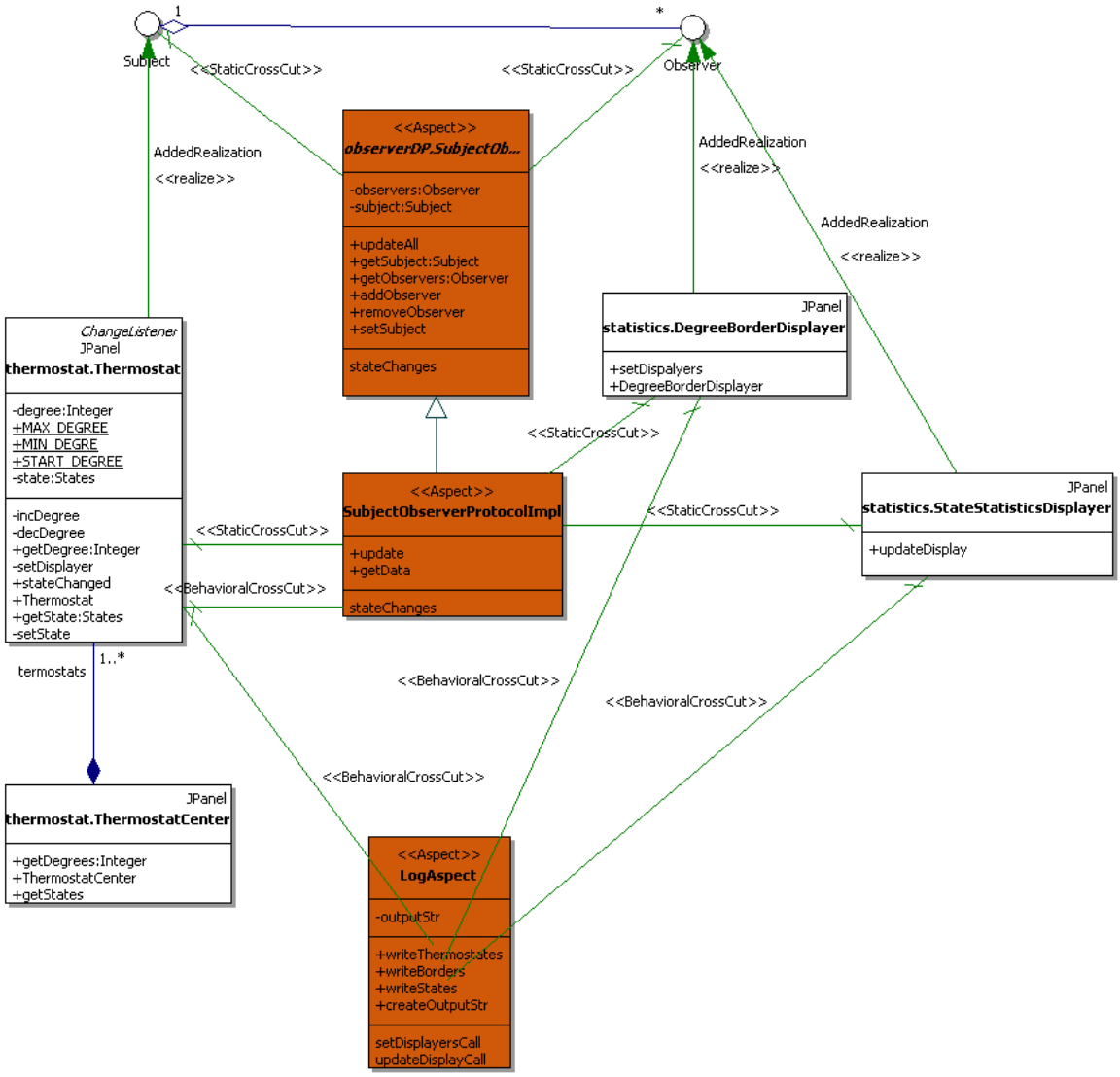
A Megfigyelő tervezési mintát átmetsző feladatként modelleztem (9 ábra). A `Subject` és `Observer` interfészek a hagyományosak; az előbbi metódusai: `addObserver`, `getObservers`, `removeObserver`, `getData`, míg az utóbbi által kért metódusok: `getSubject`, `setSubject` és `update`. A `SubjectObserverProtocol` absztrakt aspektus implementálja az interfészek metódusainak jelentős részét: a `subject` attributum az `Observer`-t bővíti és ennek segítségével implementálhatók a `setSubject` és `getSubject` metódusok, az `observers` egy `Observer` típusú objektumokól álló tömb és a `Subject` interfészt bővíti, ami felhasználásával az aspektus implementálja a `getObservers`, `addObserver`,



7. ábra. Thermostat csomag. A szürkével jelölt osztályokat a standard javax.swing csomag tartalmazza. A Thermostat egy sliderből áll, mellyel állítható a hőmérséklet, illetve egy cimkéből, mely jelzi a beállított hőmérsékletet. A MAX_DEGREE, MIN_DEGREE és START_DEGREE statikus változók, a hőmérsékleti skálát és a kiinduló hőmérsékleti értéket határozzák meg. A Thermostat-nak négy állapota lehet a hőmérséklet függvényében, az állapotok neveit a States felsorolás szolgáltatja. A stateChanged metódus a ChangeListener interfész által kért metódus, akkor hívódik meg, ha állítottunk a slideren, azaz változott a hőmérséklet. A ThermostatCenter több Thermostat-ból áll, melyekről információt szolgáltat.



8. ábra. Statistics csomag. A szürkével jelölt osztályokat a standard javax.swing csomag tartalmazza. A Statistics statikus osztály, mely az összegzéseket végzi. A StateStatisticsDisplayer illetve a DegreeBorderDisplayer megjeleníti a Statistics osztály által szolgáltatott eredményeket.



9. ábra. Osztálydiagram

`removeObserver` metódusokat (tehát ezek a metódusok az `Object` és `Subject` interfészeket átmetsző metódusok). Továbbá az aspektus deklarál egy `stateChanges` absztrakt pontszűrőt, melynek majd azokat a pontokat kell kiszűrnie, ahol a konkrét alany megváltoztatja az állapotát. Ezt a pontszűrőt használva implementálható az `updateAll after returning` típusú tanács, mely feladata, hogy az alanyhoz csatolt összes megfigyelőt frissítse az `Observer` interfészben deklarált `update()` metódusának meghívásával.

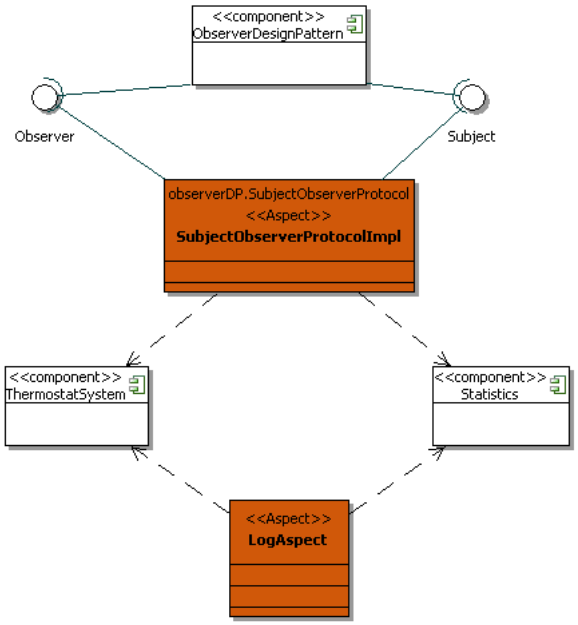
A `DegreeBorderDisplayer-Observer`, `StateStatisticsDisplayer-Observer` és `Thermostat-Subject` `AddedRealization` relációkat a `SubjectObserverProtocolImpl` aspektus deklarálja, mely a `SubjectObserverProtocol` leszármazottja. Az aspektus deklarálja az `update()` metódust, mely a `DegreeBorderDisplayer` és `StateStatistics-Displayer` osztályokat metszi át (tehát program szintjén két metódust jelent) és a `getData` `crosscuttingOperation`-t, mely a `Thermostat` osztályt átmetsző metódus. A `Subject-ObserverProtocolImpl` esetén a `stateChanges` pontszűrőt konkrétan definiálni kell: a `Thermostat` osztály `stateChanged` metodushívásait szűri ki, kontextusként a célosztályt választva ki (az alany). A célosztályra azért van szükség, mert annak a megfigyelőit kell frissíteni.

A `LogAspektus` a naplózást valósítja meg. Egy kimeneti adatfolyamot létrehozó metódust, három tanácsot és két pontszűrőt deklarál:

- a `writeThermostates` tanács a `stateChanged` pontszűrőt hasznája és bejegyzi a hőszabályzót, mely állapota megváltozott,
- a `writeBorders` tanács beírja a legkisebb és legnagyob hőmérsékletet, ahányszor a `setDisplay` metódus meghívódik, a kontextus a `setDisplay` paraméterei (a minimum és maximum érték),
- a `writeStates` tanács az állapotok-statisztikáját jegyzi be, minden `updateDisplay` metódushívás után, a kontextust ebben az esetben is a paraméter képezi.

Megjegyzem, hogy a modell nem teljes, a vizuális komponenseket egy `JFrame` keretre lehetne elhelyezni. Miután az objektumok létrehozása megtörtént, szükséges a `Thermostat` objektumaihoz, hozzáadni a megfigyelőket.

A komponensdiagram szemlélteti (10 ábra), hogy a `SubjectObserverProtocolImpl` aspektus hogyan teszi a `ThermostatSystem` és a `Statistics` komponenseket a Megfigyelő tervezési minta résztvevőivé, anélkül, hogy ezek bármilyen interfészt is implementálnának.



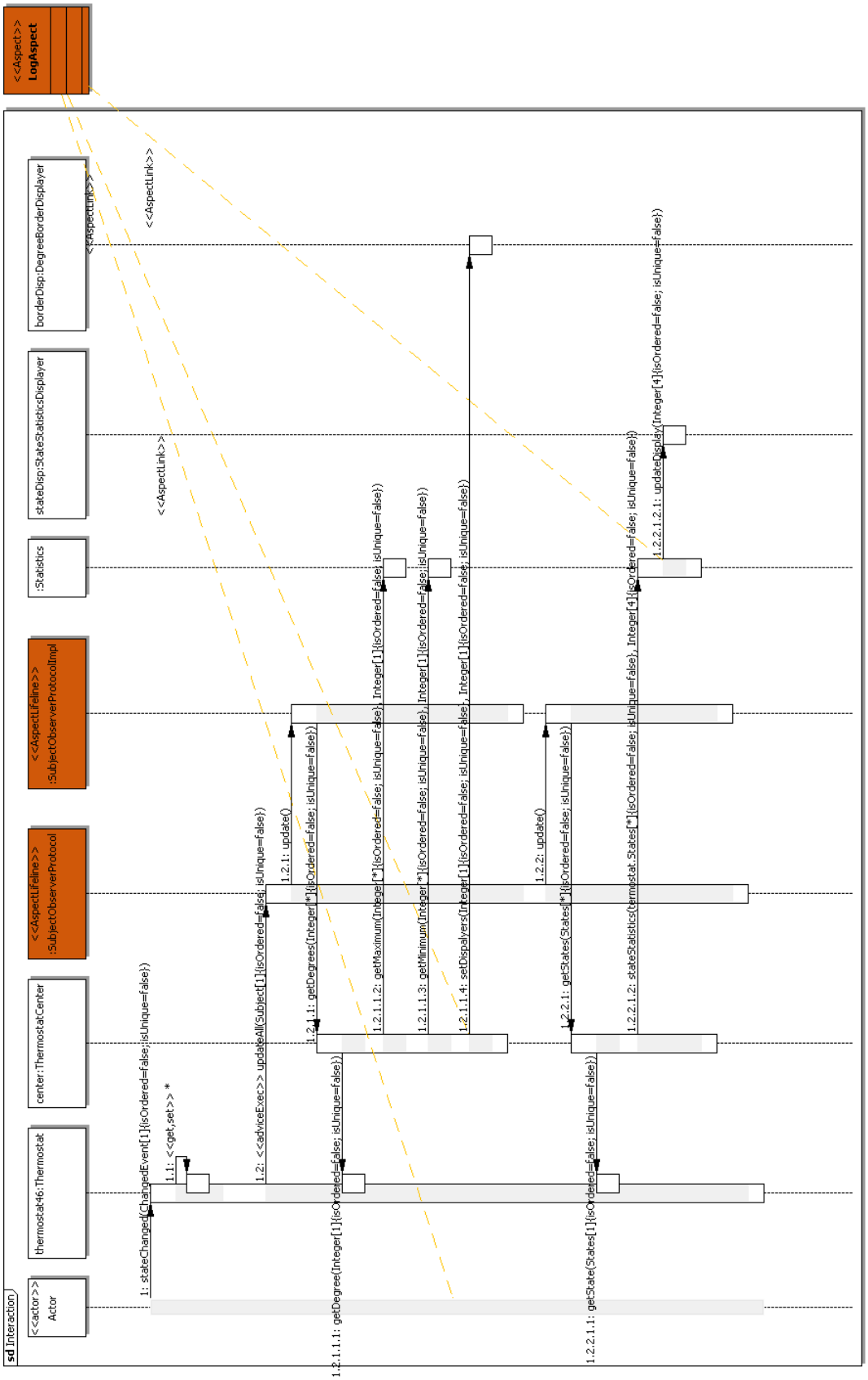
10. ábra. Komponensdiagram. Az ObserverDesignPattern az Observer és Subject interfészeket kéri, melyeket a SubjectObserverProtocolImpl aspektus implementál a ThermostatSystem és Statistics komponensek felhasználásával. A LogAspect szintén ezt a két komponenst használja.

A `LogAspect` is olyan módon avatkozik bele a két komponens működésébe, hogy azoknak nincs tudomásuk róla.

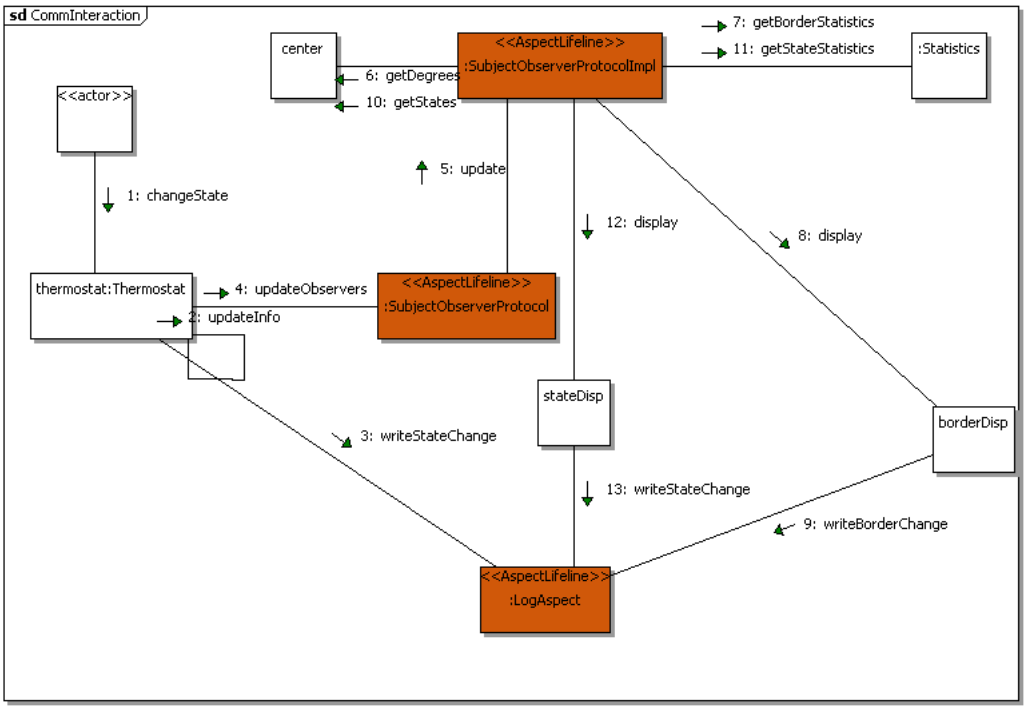
A szekvenciadiagram esetén (11 ábra) az aspektusoknak két különböző jelölésmódját láthatjuk. A `SubjectObserverProtocolImpl` és `SubjectObserverProtocol` aspektusok nagyrészt statikusan metszik a más osztályokat, ami a szekvenciadiagramon nem szemléltethető, megjelenítési módjuk ezért hasonló az objektumokéhoz, a `LogAspect` dinamikus átmetszési pontjai viszont jobban láthatók az új jelölésmóddal. Az üzenetek esetén megfigyelhető a `get`, `set`, `adviceExec` sztereotípusok használata.

A kommunikációs diagram (12 ábra) már nem felel meg az átmetszés szemléltetésére, segítségével azt ábrázolhatjuk, hogy hogyan, milyen sorrendben kommunikálnak az aspektusok és objektumok egymással.

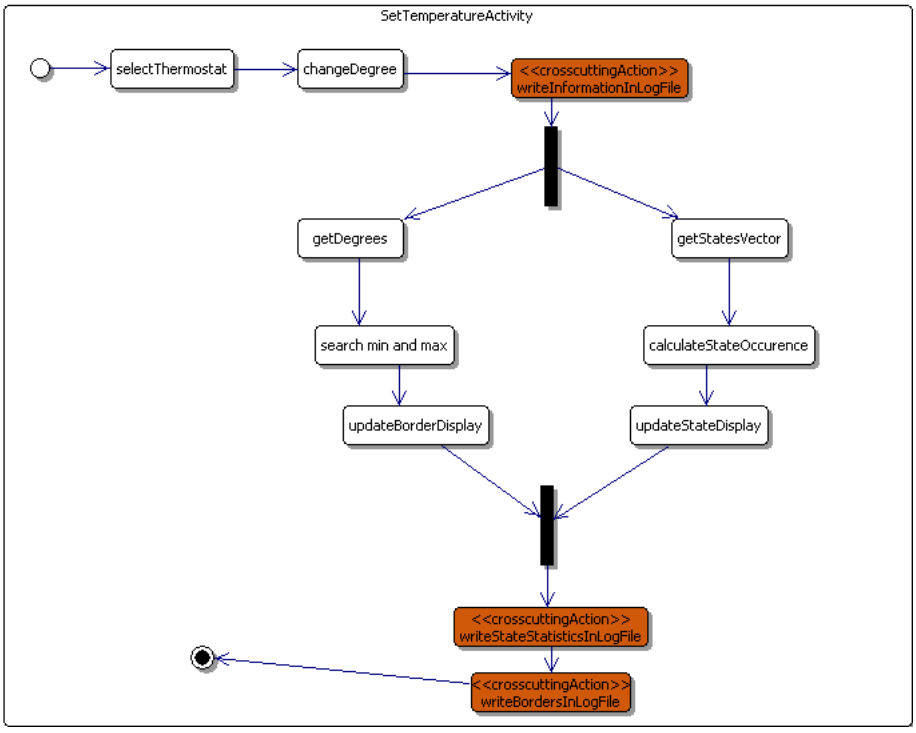
Az aktivitásdiagram (13 ábra) segítségével sikerült beazonosítani a naplózást mint átmetsző feladatot, melyet a `writeInformationInLogFile`, `writeStateStatisticsInLogFile`, `writeBordersInLogFile` átmetsző műveletek valósítanak meg. Az frissítés művelet esetén az átmetszés nem nyilvánvaló, ami azzal magyarázható, hogy végrehajtására csak egyetlen függvény hívásakor van szükség, tehát úgy is modellezhető, mint a módosításhoz tartozó művelet. Azonban figyelembe véve az osztálydiagramok esetén nyilvánvaló stati-



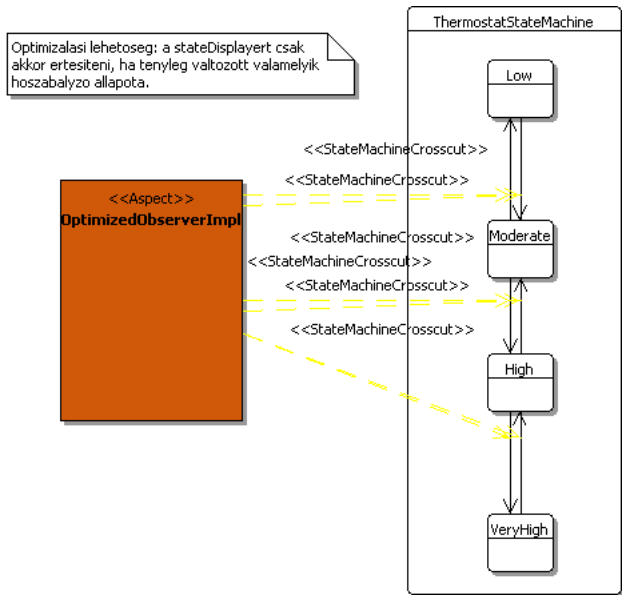
11. ábra. Szekvenciadiagram. A LogAspect apektus három programpontot választ ki, mindhárom egy-egy metódushívás, ezért a függőségek céljai Int_InvocationSpecification típusú objektumok (szürke téglalapok az életvonalakon).



12. ábra. Kommunikációs diagram



13. ábra. Aktivitásdiagram. Az állapotok összegzésének frissítése és a vele járó műveletek és a minimális és maximális érték frissítése történhet párhuzamosan is, egymástól függetlenül, a naplózási műveletek azonban csak egymást követően hajtódhatnak végre, mert azonos kimeneti adatfolyamot használnak.



14. ábra. Állapotgépdiagram. Egy hőszabályzó a Low állapotban van, ha a beállított hőmérséklet 15 fok alatti, Moderate állpotban van 15 és 30 fok között, High állapotban van 31 és 45 fok között és VeryHigh állapotban van 45 fok felett. Ezek a feltételek OCL nyelvben adhatók meg és nem jeleníti meg a diagram. Az OptimizedObserverImpl az állapotok közötti átmenetekben metszi át a Thermostat típusú objektumot.

kus átmetszést, amit a Megfigyelő tervezési minta használata eredményez, világossá válik, hogy a frissítés is egy átmetsző feladat.

Az állapotgépdiagram (14 ábra) segítségével egy optimizálási lehetőséget ábrázoltam: az állapot-statisztikát csak abban az esetben kell módosítani, ha a hőszabályzó egyik állapotból a másikba kerül. Ebben az esetben a `SubjectObserverProtocolImpl` olyan pontszűrőt kell használjon, mely az állapotátmeneteket szűri ki. Ez gyakorlatilag azokat a programpontokat jelenti, ahol a `Thermostat` objektumok `state` mezőjét változtatjuk.

4. Következtetések

Viszonylag kevés erőbefektetéssel sikerült lehetőséget teremtenem az aspektus-orientált elemek modellezésére, mely kompatibis a már létező UML modellekkel, így azok újrahasználhatók és tovább bővíthetők. A profil másik előnye, hogy az UML-jelöléseken alapszik, ezért az új elemek használatának és jelöléseinek elsajátítása egyszerű. Profilt használva bővítési eszközként a modellek tárolása a standard XMI segítségével történik.

Az UML-profil készítését támogató CASE-eszközök az UML-specifikációban nem szereplő plussz megkötéseket tesznek a profildefiniálásra, ezért a gyakorlatilag elkészített profilok nem nyújtanak annyi lehetőséget, mint amennyi elméletileg lehetséges lenne. A

Borland Together standardtól eltérő metamodellje megakadályozott abban, hogy olyan profilt készítsek, amely teljes mértékben kompatibilis az UML-specifikációval, ami leszűkíti használhatóságának körét.

Bár az esetek többségében elégségesnek bizonyult új sztereotípusok létrehozása, új metaosztályok bevezetése az aspektus-orientált modellezésre alkalmasabb és helyesebb metamodellt eredményezne, így például helyesebb volna, ha az aspektusokat a `Classifier` metaosztály leszármazottjaként modelleznénk és a pontszűrő modellezésére is teljesen új metaosztályt vezetnénk be. A nehézsúlyú bővítés megvalósítása azonban lényegesen több munkát igényel.

A modellezési nyelvek használatának legnagyobb előnye, hogy részben automatizálható a kódgenerálás, ami csökkenti a hibaforrások számát és az implementálás időtartamát. Egy további tervem a profilban definiált aspektus-orientált modellelemeknek megfelelő AspectJ-kód generálása[5]. Újrahasznosítás céljából gyakran szükség van a programkód modellé alakítására is, a projektem ezzel a funkcióval is kibővíthető.

[5]A standard elemeknek megfelelő Java kódgenerálás már meg van valósítva a Borland Togetherben, ezért elégséges csak az új modellelemeknek megfelelő kódgenerálás.

Hivatkozások

[1] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, John Irwin: *Aspect-Oriented Programming*, Proceedings European Conference on Object-Oriented Programming, vol. 1241, 220–242, 1997

[2] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm and William G. Griswold: *An Overwiev of AspectJ*, Lecture Notes in Computer Science, vol. 2072, 327–355, 2001

[3] Xerox PARC Corporation: *The AspectJ Programming Guide*, 2002-2003, http://www.eclipse.org/aspectj

[4] OMG: *Unified Modeling Language: Superstructure, version 2.0 formal/05-07-04*, 2005

[5] OMG: *Unified Modeling Language: Infrastructure, version 2.0 formal/05-07-05*, 2005

[6] OMG: *OCL 2.0, Final Adopted Specification ptc/03-10-14*, 2003

[7] Ivar Jacobson: *Use Cases and Aspects - Working Seamlessly Together*, Journal of Object Technology, vol. 2, no. 4, 7–28, 2003, http://www.jot.fm/issues/issue_2003_07/column1

[8] Siobhán Clarke: *Extending standard UML with model composition semantics*, Science of Computer Programming, vol. 44, 71–100, Inc. Elsevier North-Holland, Amsterdam, 2002

[9] Mark Basch, Arturo Sanchez: *Incorporating Aspects into the UML*, AOM workshop at International Conference on AOSD, 2003

[10] Omar Aldawud, Tzilla Elrad, Atef Bader: *UML Profile for Aspect-Oriented Software Development*, In Proceedings of Third International Workshop on Aspect-Oriented Modeling, 2003

[11] Dominik Stein, Stefan Hanenberg, Rainer Unland: *A UML-based Aspect-Oriented Design Notation For AspectJ*, Proceedings of the 1st international conference on Aspect-oriented software development, 2002

[12] Joerg Evermann: *A Meta-Level Specification and Profile for AspectJ in UML*, AOM workshop at International Conference on AOSD, 2007

[13] Thomas Cottenier, Aswin van den Berg, Tzilla Elrad: *Motorola Weavr: an add-in for Aspect-Oriented Modeling in Telelogic TAU G2*, Telelogic Americas User Group Conference, 2006