Objektumorientált tervezési alapelvek és tervezési minták döntésalapú elemzése, a döntésösszevonás elmélete és gyakorlata

Egyetemi doktori (PhD) értekezés

A szerző neve: Márien Szabolcs
A témavezető neve: Dr. Terdik György

Debreceni Egyetem
Természettudományi Doktori Tanács
Informatikai Tudományok Doktori Iskola
Debrecen, 2011.

Ezen értekezést a Debreceni Egyetem Természettudományi Doktori Tanács Informatikai Tudományok Doktori Iskola Alkalmazott információ technológia és elméleti háttere programja keretében készítettem a Debreceni Egyetem természettudományi doktori (PhD) fokozatának elnyerése céljából.

Debrecen, 2011.08.30.

……..
a jelölt aláírása

Tanúsítom, hogy Márien Szabolcs doktorjelölt 2009.- 2011. között a fent megnevezett Doktori Iskola Alkalmazott információ technológia és elméleti háttere programjának keretében irányításommal végezte munkáját. Az értekezésben foglalt eredményekhez a jelölt önálló alkotó tevékenységével meghatározóan hozzájárult. Az értekezés elfogadását javasolom.

Debrecen, 2011.08.30.

……
a témavezető aláírása

Objektumorientált tervezési alapelvek és tervezési minták döntésalapú elemzése, a döntésösszevonás elmélete és gyakorlata

Értekezés a doktori (Ph.D.) fokozat megszerzése érdekében
az informatika tudományágban

Írta: Márien Szabolcs okleveles programtervező matematikus

Készült a Debreceni Egyetem Informatikai Tudományok doktori iskolája
(„Alkalmazott információ technológia és elméleti háttere” programja) keretében

Témavezető: Dr. Terdik György

A doktori szigorlati bizottság:

elnök: ……. …….

tagok: ……. …….

……. …….

A doktori szigorlat időpontja:

Az értekezés bírálói:

……. …….

……. …….

……. …….

A bírálóbizottság:

elnök: ……. …….

tagok: ……. …….

……. …….

……. …….

……. …….

Az értekezés védésének időpontja:

Köszönetnyilvánítás

Ezúton is szeretnék köszönetet mondani mindazoknak, akik segítettek abban, hogy ez a disszertáció elkészüljön.

Elsősorban témavezetőmnek, Terdik György tanár úrnak, hogy munkámat támogatta. Ugyancsak köszönettel tartozom Arató Mátyás tanár úrnak, hogy komoly támogatást nyújtott nekem mind a tanulmányaim mind a kutatásaim során.

Végül, de nem utolsó sorban köszönöm édesanyámnak, feleségemnek, családomnak és barátaimnak, hogy támogattak, bíztattak és végig mellettem álltak.

TARTALOMJEGYZÉK

1 BEVEZETÉS 1

2 ÁTTEKINTÉS 4

2.1 AZ OBJEKTUMORIENTÁLT PROGRAMOZÁS ALAPESZKÖZEI 4
2.2 AZ OBJEKTUMORIENTÁLT TERVEZÉS ALAPELVEI 9
2.3 REFAKTORÁLÁS 15
2.4 OBJEKTUMORIENTÁLT PROGRAMOZÁSI METRIKÁK 15
2.5 A PROGRAM FORMÁLIS LEÍRÁSÁNAK MÓDSZEREI 22
2.6 TERVEZÉSI MINTÁK 24
2.6.1 Tervezési minták formalizálása 28

3 AZ OBJEKTUMORIENTÁLT PROGRAMOZÁS DÖNTÉSALAPÚ VIZSGÁLATA 30

3.1 DÖNTÉS ÉS DÖNTÉSKIEMELÉS 32
3.2 A DÖNTÉSREDUNDANCIÁK ELKERÜLÉSÉNEK ESETEI 37
3.3 DÖNTÉSÖSSZEVONÁS 37
3.4 DÖNTÉSKIEMELÉS, DÖNTÉSÖSSZEVONÁS PÉLDA 42
3.5 OBJEKTUMORIENTÁLT PARADIGMÁK KIEGÉSZÍTÉSE 43
3.6 OBJEKTUMORIENTÁLT TERVEZÉSI ALAPELVEK KIEGÉSZÍTÉSE 47
3.7 DÖNTÉSSZERKEZET ELEMZÉSE ÚJ METRIKÁKKAL 50
3.7.1 Döntésszerkezeti metrikák felhasználásának mérési eredményei 52
3.7.1.1 Döntésabsztrakciós metrika mérési eredményei 53
3.7.1.2 Döntéskiemeléssel létrejött öröklődések aránya metrika mérési eredményei 54

4 DÖNTÉSEK VIZSGÁLATA JML-LEL 56

4.1 JML SPECIFIKÁCIÓS ESZKÖZEINEK RÖVID BEMUTATÁSA 57
4.2 DÖNTÉSEK FORMALIZÁLÁSA NEM KIEMELT ÁLLAPOTBAN 58
4.3 DÖNTÉSEK FORMALIZÁLÁSA KIEMELT ÁLLAPOTBAN 61
4.4 DÖNTÉSÖSSZEVONÁS ESETEINEK VIZSGÁLATA JML-LEL 70

5 TERVEZÉSI MINTÁK DÖNTÉSALAPÚ VIZSGÁLATA 73

5.1 TERVEZÉSI MINTÁK DÖNTÉSSZERKEZETEK SZERINTI ÉRTELMEZÉSE 76
5.1.1 Létrehozási minták 76
5.1.1.1 Gyártó függvény tervezési minta 76
5.1.1.2 Elvontgyár tervezési minta 78
5.1.1.3 Építő tervezési minta 79
5.1.1.4 Prototípus tervezési minta 80
5.1.2 Szerkezeti minták 81
5.1.2.1 Illesztő tervezési minta 81
5.1.2.2 Híd tervezési minta 82
5.1.2.3 Összetétel tervezési minta 84
5.1.2.4 Díszítő tervezési minta 84
5.1.2.5 Pehelysúlyú tervezési minta 85
5.1.2.6 Helyettes tervezési minta 87
5.1.3 Viselkedési minták 88
5.1.3.1 Értelmező tervezési minta 88
5.1.3.2 Sablonfüggvény tervezési minta 89
5.1.3.3 Felelősséglánc tervezési minta 91
5.1.3.4 Parancs tervezési minta 92

5.1.3.5 Bejáró tervezési minta 92
5.1.3.6 Közvetítő tervezési minta 93
5.1.3.7 Megfigyelő tervezési minta 95
5.1.3.8 Állapot tervezési minta 98
5.1.3.9 Stratégia tervezési minta 99
5.1.3.10 Látogató tervezési minta 100
5.1.4 Tervezési minták osztályozásának új módszere 102
5.2 SABLONFÜGGVÉNY TERVEZÉSI MINTA DÖNTÉSSZERKEZET SZERINTI FELISMERÉSE AZ „EFILTER" PROJEKTEN 105
5.3 TERVEZÉSI MINTÁK JELLEMZÉSE DÖNTÉSKIEMELÉS ELŐTT ÉS UTÁN 107
5.3.1 Híd tervezési minta döntéskiemelés előtt és után 108
5.3.2 Példa – Híd tervezési minta döntéskiemelés előtt és után 109
5.3.2.1 1. Szint – Döntésösszevonások és kiemelések előtti állapot 111
5.3.2.2 2. Szint – „Személyes vagy céges vásárlás?" döntés kiemelése 113
5.3.2.3 3. Szint – „Készpénzes vagy bankkártyás fizetés?" döntés kiemelése 115
5.4 TERVEZÉSI MINTÁK DÖNTÉSALAPÚ FORMALIZÁLÁSA JML-LEL 117
5.4.1 Híd tervezési minta JML alapú formális leírása 117
5.4.1.1 Híd tervezési minta tulajdonságai döntéskiemelés előtt 118
5.4.1.2 Híd tervezési minta tulajdonságai döntéskiemelés után 124
5.4.2 Híd tervezési minta JML alapú formális leírása példán keresztül 131
5.4.2.1 Híd tervezési minta tulajdonságai döntéskiemelés előtt 131
5.4.2.2 Híd tervezési minta tulajdonságai döntéskiemelés után 139
5.4.3 Tervezési minták döntésszerkezetek szerinti JML alapú formalizálásának szabályai 152

6 KONKLÚZIÓ 154

7 ÖSSZEFOGALÓ 157

8 SUMMARY 163

9 IRODALOMJEGYZÉK 169

1 Bevezetés

Az objektumorientált programozási módszertan célja az, hogy jól strukturált, átlátható, könnyen bővíthető és karbantartható, újra felhasználható, moduláris programok szülessenek. Számos megoldás létezik annak mérésére, hogy az említett célok mennyire teljesülnek. A teljesülés méréséhez különböző metrikák használhatók, amelyek több aspektusból próbálják a strukturáltságot minősíteni.
Az objektumorientált rendszerek minőségi kérdéskörének kulcspontja a csatoltság és függőség – implementációs függőség mértéke, amely magas foka esetén a rendszer nem karbantartható, inkonzisztens modulokkal téve az adott rendszert instabillá.
A jól szervezett objektumorientált programok elérése érdekében megfogalmazásra kerültek olyan absztrakt tervezési alapelvek, melyek elsődleges célja a csatoltság, függőség – implementációs függőségek csökkentése.
Az objektumorientált tervezési alapelvekkel részben egyező célok motiválták a tervezési minták megszületését. A tervezési minták esetspecifikusan, de kellőképpen absztrakt szinten fogalmazzák meg azokat a módszereket, melyek alkalmazásával adott tervezési helyzetekben megfelelő minőségű objektumorientált felépítés jöhet létre.

Észrevételem szerint a tervezési alapelvek és tervezési minták legfontosabb célja az implementációs függőségek típusain belül a különböző döntésszerkezetek szerinti döntésredundanciák, mint egyfajta implementációs függőségek feloldása. Nem megfelelően implementált, redundáns döntések implementációs függőségeket eredményeznek, melyeket döntésszerkezeti implementációs függőségnek nevezek.
A programban lévő döntések során arról döntünk, hogy egy adott ponton milyen funkcionalitásra és/vagy adatszerkezetre van szükségünk. Amikor egy döntést definiálunk, a döntés döntéslehetőségeinek funkcionalitását és/vagy adatszerkezetét adjuk meg. A döntéslehetőségen a döntés egy lehetséges kimenetelét értjük, amely a döntéslehetőségnek megfelelő funkcionalitás és/vagy adatstruktúra érvényre jutását jelenti, illetve a döntéslehetőség teljesülését jelentő döntéspredikátumot, ami alapján eldől, hogy egy adott döntési helyen melyik döntéslehetőség jut érvényre.[30][31]
Ahhoz, hogy a döntésredundanciák feloldásának lehetőségeit megértsük, az objektumorientált módszertan alapeszközének, az öröklődésnek az értelmezését kell kiterjeszteni.
A refaktorálás, mint kód minőség javítás módszere, foglalkozik a döntés struktúrák osztály-alosztály kapcsolatokba történő kiemelésének lehetőségeivel, amellyel a döntés szerkezetek absztraktabb módon definiálhatók. [18] A döntéskiemelés során a döntés definíciót egy absztraktabb formában, azaz egy osztályhierarchiában adjuk meg, ahol a döntés „felületét” egy polimorf metódus jelenti.
Ugyanakkor nem részletezik a döntésredundanciák esetei szerinti döntésösszevonások lehetőségeit, melyet dolgozatom egyik eredményeként a döntés struktúrák refaktorálással történő optimalizálásának fő lehetőségeként értelmezek.
Az öröklődést a programban lévő döntésredundanciák feloldásának eszközeként értelmezem, amely a döntések kiemelése és összevonása után azok deklarációjára

ad egy absztraktabb keretet. Ezzel természetesen a többi objektumorientált alapfogalom is más megközelítésbe kerül. [30][31]
A programokban lévő döntésismétlődések megszüntetésére a döntések összevonásával van lehetőség, amely első lépése nem kiemelt döntések esetén azok osztályhierarchiákba történő kiemelése, amely után a döntések lehetőségeihez tartozó adattartalmak és funkcionalitások egy osztály alosztályaiban kapnak keretet.
Kiemelve a döntéseket és a döntéslehetőségeket egy osztályba és annak alosztályaiba, a döntésösszevonás szabályai szerinti döntésösszevonásokat elvégezve lehetővé válik, hogy a döntéseket egy helyen meghozva, annak eredményét a kiemelt és összevont döntések szülőosztályába és annak alosztályaiba „zárva" (példányosítás) archiváljuk. A létrejövő objektumot egy referencia változóval hivatkozzuk, amely típusa a szülőosztály. Az adott döntés további döntési pontjain már a meghozott, archivált döntés eredményét – szülőosztály típusú referencia típusú változót – használhatjuk fel úgy, hogy nem kell figyelni arra, hogy a döntés során mely döntéslehetőség érvényesült, azaz melyik alosztályt példányosítottuk, hiszen a szülőosztály típusával hivatkozva a későbbi felhasználások során rejtve, bezárva marad, mivel a döntés felületét egy polimorf metódus adja. [30][31]
A döntéseknek két állapotát különböztetjük meg attól függően, hogy az adott döntés definíciót, azaz a döntéslehetőségek adatstruktúráit, metodológiáját egy metódusban adták-e meg, vagy már kiemelten, egy osztályban és annak alosztályaiban. [30][31]
Megfogalmazok két új tervezési alapelvet. Az első alapelv – „Öröklődést döntésredundancia megszüntetésére használjunk!" – egyértelműen meghatározza azon eseteket, amikor az öröklődés, mint objektumorientált eszköz használata az objektum összetétellel szemben indokolt lehet. Mivel a már meglévő tervezési alapelvek egy kritikus pontja annak eldöntése, hogy az öröklődés használata mely esetekben indokolt, ezért az új elv lényeges kiegészítést jelent a már meglévő alapelvek tekintetében. A döntésredundancia csökkentése, mint új cél elérése érdekében megfogalmazott másik új alapelv: „A döntésredundanciát kerüljük!".
Az új elképzelésnek megfelelően az objektumorientált alapeszközök és a már meglévő tervezési alapelvek értelmezését is kiegészítem, amely alapján a tervezési alapelvekre épülő tervezési minták céljai, leírásai is változnak.
Olyan új objektumorientált metrikákat vezetek be, melyekkel lehetőség van a programban szereplő döntésredundanciák mértékének megállapítására, illetve, melyekkel a bevezetett tervezési alapelvek teljesülése mérhető. („Döntésabsztrakciós metrika", „Döntéskiemeléssel létrejött öröklődések aránya", „Ekvivalens döntésesetek aránya", „Döntéspredikátumában eltérő, egyéb viselkedésében egyező döntésesetek aránya", „Ekvivalens döntéspredikátumú döntésesetek aránya" metrikák)
A tervezési minták új értelmezés szerinti célja: a jól strukturáltság elérése a döntésismétlődések megszüntetésével. Tehát a tervezési minták a döntéskiemelésekre adnak recepteket, vagyis arra, hogy a különböző strukturált döntési helyzetekben a döntésismétlődés megszüntetése miként biztosítható. [30][31] A tervezési minták osztályozhatók aszerint, hogy milyen döntésszerkezetek esetén adják meg az optimális döntésdeklarációkat, illetve, hogy a döntésarchiválás mely módját támogatják.

A tervezési minták alkalmazásának egy új irányát vehetjük észre, miszerint a döntések nem kiemelt állapota esetén is észlelhetők azok a döntéskapcsolatok, amelyek az adott tervezési mintára jellemzőek, így a tervezési minták nem csak a jelenlegi formájukban ismerhetők fel, hanem a döntések nem kiemelt állapota esetén a tervezési minták „lenyomatai” is észlelhetők. [30][31]
Ez új lehetőséget ad annak tisztázására, hogy milyen esetekben lehetséges egy adott tervezési minta felhasználása, hiszen egy tervezési minta leírásakor nem egy optimális megoldás paramétereit kell, hogy megadjuk, hanem egy olyan tervezési helyzet körülményeit, amikor egy adott tervezési minta felhasználásával egy optimálisabb struktúra kialakítása érhető el. Tehát a tervezési minták bevezetésének lehetősége sokkal érthetőbbé válik, mivel konkrét megoldástól független jellemzők, a döntésszerkezet szerint határozható meg egy tervezési minta alkalmazhatósága.
Abban az esetben, ha a döntések elemzése automatizált eszközökkel is megtehető, akkor lehetőség nyílhat a tervezési minta alkalmazás szükségességének automatikus felismerésére.
Hogy az elképzelésnek megfelelően a strukturáltságot, azaz a jól strukturált program irányelveinek teljesülését elemezni tudjuk, szükséges egy olyan formalizációs eszköz, amellyel a döntés definíciók vizsgálhatók. Erre a Java programok viselkedés specifikációs nyelvét a JML-t [26] használom [31], amely lehetőséget ad a döntés lehetőségeinek adatstruktúráját, metodológiáját logikai kifejezésekre alapozva specifikálni. Az így formalizált döntések vizsgálata a jól strukturáltság szabályainak való megfelelés szempontjából már lehetséges.
A JML alapú döntés specifikációra alapozva a tervezési mintákat jellemző döntésszerkezet is modellezhető. A „Híd” tervezési mintában található döntésszerkezet deklarációk különböző állapotainak formális leírását külön-külön adom meg, amely alapján megfogalmazom azokat az általános szabályokat, amelyek a JML-es döntésszerkezet szerinti tervezési minta formalizálás alapjai lehetnek.

Elképzelésem szerint tehát az objektumorientált módszertan tervezési alapelveinek és alapeszközeinek célja a döntésismétlődések megszüntetése, amelyeket a döntések kiemelésével tehetünk meg. A tervezési minták a különböző tervezési helyzeteknek megfelelő döntésszerkezetekre adnak optimális osztály és objektum szerkezeti megoldásokat, vagyis a tervezési minták jellemzését érdemes döntésszerkezetük alapján megtennünk. [30][31]

2 Áttekintés

A dolgozatban felvázolt elképzelés elsődlegese célja az objektumorientált programozási és tervezési módszertan olyan új gondolatokkal való kiegészítése, melyekkel az objektumorientált programok strukturáltsága, tervezési minősége javítható.
Habár a minőséget nehéz kifejezni, a minőségi termékeknek egyértelmű hatásai vannak: könnyű velük dolgozni, könnyen megérthetők, könnyen átalakíthatók, kevés hibát észlelünk használatuk során.
Az informatikában a minőségi programok, azaz termékek születését próbáljuk elősegíteni. Ahhoz, hogy tudjunk viszonyítani, tudnunk kell mérni a minőséget. A minőség mérésére a szakma a programozási metrikákat használja, amellyel számos aspektusból lehetséges a kódminőség mérése.
A programminőség, strukturáltság javításának céljából számos korábbi objektumorientált tervezési alapelv került megfogalmazásra, mely a minőségjavulást a programokban lévő csatoltságok csökkentésével és függőségek feloldásával érte el.
Ezen tervezési alapelvek céljaival összhangban születtek meg a tervezési minták, melyek az alapelvek szerinti strukturáltság javulását esetspecifikusan, de kellőképpen absztrakt szinten fogalmazzák meg, hogy felhasználásuk esettípusra adoptálható legyen.
A tervezési minták a felhalmozott tervezési tapasztalatokat újra hasznosíthatóvá teszik, elősegítve minőségi rendszertervek és azok alapján programok születését.

A dolgozatban olyan új objektumorientált programozási és tervezési alapelveket vezetek be, melyekkel a korábbi alapelvek szerinti célok: csatoltság csökkentés, függőségfeloldás teljesülése új szempontok szerint elősegíthető úgy, hogy közben a korábbi tervezési alapelvek értelmezése is kiterjeszthető.
Az elképzelésre alapozva új metrikák bevezetését kezdeményezem, amellyel a koncepció szerinti strukturáltság mérése megvalósítható.
Fontos kiemelni, hogy a korábbi tervezési alapelvek mellett a tervezési minták értelmezése, leírása is új megközelítésbe kerül, amely bemutatása a dolgozat fontos részét képezi.

Az alfejezetekben az objektumorientált tervezési alapelveket, az azokra épülő tervezési mintákat és a minőség mérésére használt metrikákat mutatom be.

2.1 Az objektumorientált programozás alapeszközei

Az objektumorientált programozás egy programozási módszertan, mellyel megvalósuló programok az objektumok együttműködését szervezik úgy, hogy azok mindegyike valamely osztály egyede. Az osztályok egy olyan hierarchiának részei, amely öröklődési kapcsolatokkal épül fel. [5]
Az objektumorientált nyelveknek a következő eszközökkel kell rendelkezni: absztrakt adattípus kezelés, öröklés, dinamikus/kései kötés. [45]
Ahhoz, hogy jobban meg tudjuk érteni az objektumorientált tervezés és programozás alapelveit, meg kell fogalmaznunk mi az absztrakció: Amikor

deklaráljuk egy egyed azon tulajdonságait, amely az összes többi egyedtől megkülönbözteti, amellyel az adott egyed viselkedésének határait a felhasználás szemszögéből kijelöljük, absztrakcióról beszélünk (Grady Booch)[48], azaz meghatározzuk azt az interfészt, amelyen keresztül szolgáltatásai elérhetők.

Az objektumorientált programozás alapfogalmaiként az egységbezárást [5][11][39], öröklődést [5][11][39][41], polimorfizmust [5][11][39], üzenetküldést [11][41] említhetjük.

Objektumokba-osztályokba szervezés:
Az objektumorientált rendszerfejlesztés során kritikus, a rendszer tulajdonságainak-viselkedéseinek objektumokba-osztályokba szervezése, amely során számos – néha egymással versengő – tényezőt kell figyelembe vennünk, amelyek a következők: egységbezárás, finomság – részletezettség, függőségek, rugalmasság, teljesítmény, továbbfejleszthetőség – karbantarthatóság, újrahasznosíthatóság. [15]
Az objektumorientált tervezési módszerek számos megközelítést ismertetnek az objektumok – osztályok körbehatárolására. Az egyik lehetőség szerint feladatokat kell meghatározni, amelyekhez főneveket és igéket társítva egy adott osztály attribútumai és metódusai kialakíthatók.[15] Lehetőség van a rendszer elemeinek együttműködéséből kiindulva a felelősségi körök meghatározásával az osztályok behatárolására. További mód a rendszer célterületének elemzésével követelmény specifikálás közben a követelmények realizálásával felfedezett osztályok, objektumok szerint megoldani a rendszer osztályokba szervezését. (RUP, azaz Rational Unified Process [23])
Az osztályokba szervezés során szükséges a megfelelő absztrakciós szint biztosítása, amely a későbbi karbantartást, kiegészítést megkönnyíti. A tervezési minták segítenek a megfelelően elvonatkoztatott megoldások megtalálásában.
Lényeges kérdése az objektumorientáltságnak az objektumok felületeinek meghatározása, amelyen keresztül az adott objektum viselkedése, állapota elérhető. Objektum felületen értjük az adott objektumnak küldhető üzenetek halmazát, más terminológia szerint azon metódusok halmazát, amelyen keresztül az adott objektumhoz hozzáfér a környezete.[15]
Tehát az objektumok felülete az, amelyen keresztül a külvilággal az objektum kapcsolatban van. Amikor fel akarunk használni egy objektumot, a megvalósításról semmit nem tudunk, vagyis lehetséges, hogy két objektumnak teljesen egyezik a felülete, de a felület megvalósítása eltér. Az absztrakt adattípusoknál taglalt adat elrejtésből, bezárásból következő tulajdonsága ez az absztrakt adattípusokból kifejlődő objektumorientált rendszereknek, amely nagyfokú rugalmasságot biztosít az így meghatározott modulok felhasználásakor.
A polimorfizmusból fakadó kései kötés fogalomköre erre a tulajdonságra épül, amelyre lentebb részletesebben kitérek.

Példányosítás:
Egy osztály példányosítása során az osztály definíció alapján az objektum belső állapotának tárolásához szükséges tárhely lefoglalásra kerül, illetve a műveletek hozzárendelődnek. [15] Tehát a példány egy absztrakció konkrét megjelenése,

amelyre az osztályban definiált metódusok alkalmazhatók, illetve van állapota, amellyel e metódusok hatásait képes tárolni. [6]

Osztály, felület, alosztály, altípus, öröklés:
Beszélhetünk osztályról és típusról [15]. Egy osztállyal reprezentálhatjuk az objektumok egy halmazát, melyeket ugyanazon attribútumokkal írhatunk le, illetve viselkedésük, kapcsolataik egyeznek.[6] Egy objektum osztálya megadja annak belső állapotleírását, illetve a rajta értelmezett műveleteket. Egy típus az adott objektum felületére, azaz a rajta értelmezett metódusokra utal.
Az objektumorientált tervezés kulcspontja az öröklődés. Az öröklődés jelentése, hogy egy már meglévő osztályban definiált adatstruktúrákat, metódusokat öröklik az adott osztályból származó alosztályok úgy, hogy kiegészíthetik azokat új adatstruktúrákkal, metódusokkal [37][41], illetve az öröklött tulajdonságokat felül is definiálhatják. Az öröklődés teszi logikusan átláthatóvá a tervet, és megszabadítja azt a kód redundanciáktól
Öröklésről beszélünk, mikor típusok, osztályok közös tulajdonságait kiemelve, azokat egy közös új típus-osztály tag változóiként és metódusaiként deklaráljuk, melyeket örökölni fognak a speciális kiegészítéseket adó altípusok, alosztályok. [6]
Altípus, azaz felületöröklés esetén azt adjuk meg, hogy egy objektumot mikor használhatunk egy másik helyett.[15]
Osztályöröklés, alosztály esetén a szülőosztály szolgáltatásait tudjuk bővíteni, lehetőséget adva arra, hogy azok felé közös felületet biztosítsunk, ami a többalakúság alapja. Ha az osztályöröklés során az alosztály csak felülírhatja a műveleteket vagy újakat határozhat meg, de a szülőosztály műveleteit nem tudja elrejteni, akkor minden alosztály objektuma tudja fogadni a szülőosztálynak szánt kérelmeket, azaz a szülőosztály felületén értelmezett műveleteket, tehát az alosztály egyben altípus is.[15][45]
Osztály hierarchiák, azaz öröklődés esetén „is a” kapcsolatról beszélünk.
Az öröklés statikus megoldást nyújt a felületek kiegészítésére, működésének részleges felülírására. A kód újrahasznosításának kifinomult eszköze, mivel kiegészítheti, módosíthatja a szülőosztály viselkedését úgy, hogy utána ugyanazon felületen keresztül elérhetők annak szolgáltatásai. Azonban hátrány, hogy ez a statikus függőség nem oldható fel, azaz a szülőosztály által nyújtott megvalósítások futásidőben nem változtathatók meg. Szintén kritikus problémát jelent, hogy a szülőosztály megvalósításától való függés a szülőosztály változása esetén kockázatot jelenthet. Ennek megoldása az lehet, hogy csak absztrakt osztályoktól öröklünk, amelyek nem vagy csak részlegesen rendelkeznek megvalósítással.
A többszörös öröklődés egy olyan osztályok közötti kapcsolat, mellyel egy osztály az adatstruktúráját és/vagy viselkedését meg tudja osztani más osztályokkal. Tehát ha egy alosztálynak két szülőosztálya van, akkor többszörös öröklésről beszélünk.

Egységbezárás:
Az egységbezárás – encapsulation – azt jelenti, hogy az adatstruktúrákat és a ráépülő metodológiát együtt definiáljuk összezárva, egységbe zárva azokat. Az így egységbezárt adatstruktúra és metodológia definiálására adnak keretet az osztályok, amelyek egyes példányait nevezzük objektumoknak. Az egységbezárás segítséget

ad a modularizált felépítéshez, ami lehetővé teszi, hogy ha egy objektum metodológiája változik, ne vonjon maga után változásokat az objektum környezetén.[11] Az egységbezárással lehetővé válik a bezárás, információ elrejtés: „information hiding”, ami azt jelenti, hogy egy programegység egy meghatározott interfészen keresztül nyújt szolgáltatásokat.[45]

Polimorfizmus:
A többalakúságon – polimorfizmuson azt értjük, hogy egy referencia által mutatott objektum típusa egy osztályhierarchián belül futásidőben dől el, azaz „többalakú”. Ennek megfelelően egy osztályban definiált metódusok az alosztályokban felüldefiniálhatók, így csak futásidőben kerül meghatározásra (kései kötés, „Late-Binding”), hogy ténylegesen melyik kód kapja meg a vezérlést [39]. Tehát polimorfizmus esetén egy objektumnak több típusa is lehet, és az ősosztály által meghatározott felület, metódus, kérés az alosztály által felüldefiniált. Lehetséges, hogy az adott kérést különböző objektumok (szülő vagy alosztály objektuma) kapják meg, azaz ugyanarra a kérésre más-más megvalósítás fog érvényre jutni.
A kései kötés más terminológia szerint azt jelenti, hogy egy objektum más objektumoknak (egy osztály és alosztályainak egyedei) ugyanazon üzenetet küldi, különböző kód végrehajtását eredményezi.[11] Az üzenetküldésen (Smalltalk értelmezésben üzenetküldés, egyébként pl. C++ terminológiában metódushívás [11]) azt értjük, amikor egy objektum egy másikat egy metódus végrehajtására „ösztönöz”. [41] A kései kötés lehetőségének megléte azonban programnyelv függő (illetve lehet opcionális - C++), mivel túl nagy az erőforrásigénye, habár jelentősen csökkenti a kód karbantarthatóságát.[37] A polimorfizmus növeli az újrahasználhatóságot, mivel lehetővé teszi azt, hogy olyan alosztályok és azok példányai érvényesüljenek, melyek később kerülnek bele a rendszerbe.[37]
Megvalósítása úgynevezett polimorf referenciákkal lehetséges, mellyel az osztályhierarchiában felülírt metódust hívhatunk. A polimorf referencia típusa a szülőosztály vagy interfész, amely legalább az adott polimorf metódus szignatúráját deklarálja. A polimorf referencia mutathat a szülőosztályra vagy valamelyik alosztályára úgy, hogy csak dinamikusan dől el a hivatkozott objektum tényleges osztálya. Eszerint dinamikusan dől el, hogy a polimorf metódushívás melyik osztály metódusához köthető. Ilyenkor dinamikus kötésről beszélünk. [45]
Egy adott kérés esetén lehetséges tehát, hogy a hozzá tartozó művelet meghatározása futás időben történjen, amit kései kötésnek nevezünk. Tehát a felhasználáskor a kérelmező egy adott típusú felületen keresztül szólít meg egy objektumot, ahol minden olyan objektum felhasználható, amely az adott felülettel rendelkezik úgy, hogy az objektum típusa „rejtve” marad, azaz a tényleges megvalósítás nem ismert, csak a felület. Ez azt jelenti, hogy az azonos felülettel rendelkező objektumok egymással felcserélhetők, amit többalakúságnak – polimorfizmusnak nevezünk. A felcserélhetőség garantálására született a „Programozzunk felületre megvalósítás helyett” elv.

Objektum összetétel, aggregáció:
Az objektumorientált rendszerek felépítése során a két alapvető felépítési eszköz az öröklés és az objektum összetétel. Az objektum összetételt nem létezésfüggő

kapcsolatok esetén aggregációval/képviselettel valósítjuk meg. Az öröklést fehér dobozos újrahasznosításnak nevezzük [15], amely arra utal, hogy örökléskor az alosztály részlegesen hozzáfér a szülő osztály belső szerkezetéhez, viselkedéséhez. Az aggregáció/képviselet, objektum-összetétel során az objektumok együttműködésének eredményeként fog létrejönni egy új szolgáltatás. Az aggregációval/képviselettel, objektum-összetétellel létrejövő megoldást nevezzük fekete dobozos újrahasznosításnak, mivel az együttműködő objektumok belső szerkezete egymás számára láthatatlan.
Az aggregáció/képviselet, objektum-összetétel – az örökléssel szemben – dinamikus, futás időben létrejövő kapcsolatokon alapszik. Aggregáció/képviselet, objektum-összetétel során az osztályok együttműködése azok felületein keresztül történhet, ahol a megvalósításuk rejtve marad, azaz a bezárás elve nem sérül, vagyis az egységes felületű objektumok lecserélhetők az egyes felhasználási helyeken, tehát a megvalósítási függőség [15] csökken. A fekete dobozos újrahasznosításnak megkülönböztetjük két változatát: aggregációt/képviseletet: „aggregation” és objektum összetételt: „composite”, ahol az összetétel esetén létezésfüggőség van a tartalmazó és tartalmazott között, azaz a tartalmazott objektum csak akkor és addig létezhet, amíg a tartalmazó objektum létezik. Aggregáció/képviselet esetén nincs ilyen függés.
Objektum összetételek – képviseletek/aggregációk esetén „part of” kapcsolatokról beszélünk.
Az aggregáció/képviselet esetén úgy érünk el az örökléshez hasonló működést, hogy egy elvárt működést egy külön osztályban valósítunk meg, és annak viselkedését egy referencián keresztül érjük el úgy, hogy a szolgáltatás elérésekor átadjuk a szolgáltatást kérő objektumot is, hogy a szolgáltatást nyújtó objektuma is elérhető legyen az öröklés esetén a „this” kulcsszóval megvalósuló önhivatkozáshoz hasonlóan. Az aggregációval/képviselettel megvalósuló objektum és osztályszerkezetek esetén a viselkedés dinamikusan könnyen változtatható. Ugyanakkor meg kell jegyezni, hogy az aggregáció/képviselet vagy objektum-összetétel alkalmazása nem vezethet ahhoz, hogy az örökléseket a szükséges helyeken is kicseréljük.

Paraméterezhető-generikus-általános programegységek:
Paraméterezhető-generikus-általános programegységek az objektumorientált rendszerek újrahasznosításának az objektumorientált megoldások szerves részét nem képező eszközei. Ugyanakkor a generikus programegységek növelik az újrahasználhatóságot, azaz a produktivitást, hiszen használatával elkerülhető, hogy ugyanazon algoritmust különböző adattípusokra külön-külön kelljen elkészíteni.[45] Például nem szükséges különböző rendező alprogramokat írni, ha különböző elemeket tartalmazó tömböket akarunk rendezni.
A parametrikus polimorfizmus fogalma szerint az alprogram egy típust vár generikus paraméterként, amely meghatározza az alprogram meghívásakor a típust.[45]
Az ADA és C++ után azoktól részben eltérő megközelítéssel, de a Java 5-ben is bevezetésre kerültek a generikus programegységek.

Generikus osztály vagy metódus meghatározása során nem szükséges specifikálnunk az általuk felhasznált minden típust, azokat felhasználásuk során, vagyis az osztály típusú változó deklarálásakor vagy az adott metódus meghívásakor adjuk meg.

2.2 Az objektumorientált tervezés alapelvei

Az objektumorientált rendszerek minőségi kérdéskörének kulcspontja a csatoltság és függőség mértéke.
Csatoltságon az egyes rendszerkomponensek kapcsolatainak erősségét értjük. Két komponens közötti csatoltság növekszik, ha egyik a másik valamilyen mezőjére utal, metódusát hívja vagy egy metódus visszatérési típusa a másik komponens típusa. A csatoltság legerősebb formája az öröklés. Laza csatolás esetén az osztályok módosítása nem eredményez jelentős változtatáskényszert a többi osztályban.
Az implementációs függőségek a csatoltság következményei, amelyek növelik az instabilitás kockázatát, csökkentik a karbantarthatóságot.
Az implementációs függőség csökkentése kritikus, mivel hozzájárul a karbantarthatóság, újrafelhasználhatóság tekintetében kiemelt fontosságú laza csatolás kialakításához.
Laza csatolással elősegíthető autonóm rendszerkomponensek kiépítése, amelyek cserélhetősége, karbantarthatósága könnyebben megoldható. Laza csatolás nélkül a függőségek átláthatatlan rendszert eredményeznek, amelyek a karbantarthatóságot és későbbi felhasználhatóságot teszik lehetetlenné.
A csatoltság és függőségek feloldása, az objektumorientált szoftverek minőségének javítása érdekében kerültek megfogalmazásra az objektumorientált tervezési elvek. Az objektumorientált tervezési elvek olyan absztrakt elveket fogalmaznak meg, amelyek figyelembe vételével az említett tervezési problémák elkerülhetők.
A rendszerkomponensek közötti laza csatolások elérésének támogatása az egyik fő célja számos jelenlegi keretrendszernek, amely a jelenlegi objektumorientált rendszerek, megjelent konkrét keretrendszerek (JBoss Seam [19], Spring [49], stb…) egyik legfőbb célkitűzése.

Programozzunk felületre megvalósítás helyett [15][48]:
Az elvont osztályokkal különböző típusú objektumoknak közös felületet biztosítva megoldható, hogy az adott objektum típusának, vagyis a konkrét megvalósítás ismerete nélkül csak a felület ismerete alapján használjunk fel objektumokat elrejtve azok tényleges megvalósítását.
Ezzel biztosítható az alrendszerek közötti implementációs függőség csökkentése, ami kritikus az objektumorientált rendszer újrahasználhatósága szempontjából. A „programozzunk felületre megvalósítás helyett” alapelv szerint, próbáljuk az elvárt működést absztrakt, elvont osztályokban megfogalmazni ahelyett, hogy a szükséges absztrakciós lehetőségek kihasználása helyett inkább számos felesleges függőséget építenénk be alosztályainkba.[15]
Az absztrakt viselkedés megadásának eszközei az interfészek, melyekkel megadható, hogy egy objektum egy másikról mit tud a hívott metódus

szolgáltatásainak elérésekor. Egy objektumnak lehet több interfésze, melyekkel különböző típusú szolgáltatásokat publikál. Tehát egy objektumnak lehet több interfésze, ugyanakkor több különböző objektumnak is lehet azonos interfésze is. [48]
Az alosztályok, altípusok abban különböznek, hogy alosztályok esetén osztályöröklésről beszélhetünk, azonban altípusok esetén interfész öröklésről, azaz altípusos viselkedés alkalmazásának eszköze az interfészek használata, amely elősegíti a LSP teljesülését.
Az interfész használatának egyértelmű előnye az, hogy felhasznált osztály interfészen keresztül történő felhasználásakor az implementáció bezárása egyértelműen megvalósul, amely miatt a megvalósítás könnyen, rugalmasan cserélhető, azaz a laza csatolás megvalósulását segíti, ugyanakkor a felépítés komplexitását növeli. [48]

Liskov féle helyettesíthetőség elve – LSP [15][29][44][45][48]:
Egy objektumnak több típusa is lehet illetve különböző objektumok is osztozhatnak egy típuson, melyek felületeinek közös része kell, hogy legyen a közös típus által meghatározott felület. Egy felület más felületeket is tartalmazhat részhalmazként.
Egy típus akkor altípusa egy másiknak, ha felülete örökli a másik felületét [15], azaz viselkedésileg ekvivalens az alosztály egy egyede a szülő osztály egyedével, ami azt jelenti, hogy ahol a szülő osztály típusú objektum szerepel, az helyettesíthető alosztály típusú objektummal. [45]
Egy alosztály akkor altípusa egy szülőosztálynak, ha a szülő osztályt csak metódusokkal és tag változókkal egészíti ki, illetve felül is írhatja az örökölt metódusokat úgy, hogy az bárhol helyettesítheti a felülírt metódust. Az altípus nem engedi, hogy a szülőosztálynak legyen olyan adattagja, metódusa, ami az alosztály által nem látható.[45]
A Liskov féle altípus fogalma:
Tegyük fel, hogy az f(x) tulajdonság bizonyítható minden T típusú x objektumra. Ekkor f(y)-nak igaznak kell lennie minden olyan S típusú y objektumra, ahol S a T altípusa. [29]
Egy T típus későbbiekben használt C viselkedési szerződésének értelmezése a következő: [44]

- A típus által elérhető függvények viselkedését megadó elő- és utófeltételek, illetve a függvények állapotra gyakorolt hatását deklaráló keretfeltételek.
- Az osztály tagváltozói által elérhető értékeket szemléltető invariánsok (objektum állapotait szűkítő állítások) és „history contraint”-ek (kezdő és végállapotok közötti összefüggések, melyek az állapotátmenetekre adnak megkötéseket).

Az altípus és helyettesíthetőség definícióját a viselkedési szerződés fogalmát felhasználva a következők szerint tették egyértelművé: [44]
Legyen T egy C szerződést teljesítő típus, miszerint minden T típusú o_1 objektum teljesíti a C_{o_1} szerződést. Ha minden T típusú o_1 objektumhoz létezik olyan S típusú o_2 objektum, ami szintén teljesíti a C_{o_1} szerződést, akkor minden olyan P programra, amely tartalmazza a C szerződést igaz, hogy az általa teljesülő

szerződésekre, azaz a program viselkedésére nincs hatással az, ha az o_1 objektumokat o_2 objektumokra cseréljük. Ez azt jelenti, hogy az S típus a T altípusa.[44]
Az előbbi állítást azzal egészítették ki, hogy az alosztály által specifikált szerződés finomíthatja, specializálhatja a szülő osztály által teljesített szerződést, ami nem sérti az altípus definícióját és a helyettesíthetőség elvét. Ez a viselkedési szerződés tekintetében a következőket jelenti: [21][44]

- Az altípusban az előfeltételek nem erősebbek, mint a szülőben.
- Az altípusban az utófeltételek nem gyengébbek, mint a szülőben.
- Az ős mezői felett az altípus invariánsa nem gyengébb, mint a szülőé.
- Az altípus betartja a szülő történeti megszorításait, azaz az újonnan bevezetett metódusok nem tesznek lehetővé olyan állapot átmenetet, amely az ősben nem volt lehetséges.

Objektum összetétel használata öröklés helyett, ahol lehet [15][48]:
Látni kell, hogy az objektum összetételek és öröklések használatával kapcsolatosan kritikus annak meghatározása, hogy mikor melyik konstrukció használata optimálisabb az újrahasználhatóság, karbantarthatóság szempontjából. A cél, hogy a két eszköz felhasználásával egy harmonikus osztály és objektum szerkezet keletkezzen. Az öröklés egy nagyon erős kötést jelent két osztály között, melyet használni csak indokolt esetben érdemes.
Az objektum összetétel és öröklés használatának előnyei-hátrányai: [48]
Objektum összetétel használatával az újrahasználhatóság nő, azaz objektum összetétellel egy tartalmazó objektum számára dinamikusan új funkcionalitást „szállíthatunk” úgy, hogy a tartalmazott objektum szolgáltatásai csak az interfészén keresztül érhetők el. Ezzel a tartalmazott objektum szolgáltatásainak megvalósítása rejtve marad – „fekete dobozos” újrafelhasználás, ami csökkenti az implementációs függőséget. Ugyanakkor hátránya az, hogy a túl sok objektum túl bonyolult kapcsolati rendszerrel rendelkező rendszert eredményezhet.
Az öröklésnél viszont lehetőség van egy már meglévő viselkedést újra felhasználni úgy, hogy annak a működését egy alosztályában a szükséges módosításokkal egészítjük ki. A szülőosztály tartalmazni fogja az alosztályok közös működését, illetve felületet ad a dinamikus több alakú működésnek. Hátránya, hogy az alosztály a szülőosztály viselkedését nem csak egy interfészen keresztül érheti el, azaz sérül az információ elrejtés, „information hiding”, azaz a bezárás elve. Tehát ebben az esetben „fehér dobozos” újrafelhasználásról beszélhetünk, mivel a szülőosztály belső működéséhez gyakran hozzáfér az alosztály. Ez implementációs függőséget eredményez, ami miatt, ha a szülőosztály változik, az alosztályokat is gyakran változtatni kell. Az objektum összetétellel szemben nagy hátránya, hogy az öröklés esetén a viselkedés nem cserélhető futásidőben.

Vezérlés inverzió elve („Inversion Of Control”, „Hollywood Principle”): [14]
„Ne hívj, majd mi hívunk!”
Az „Inversion of Control” a keretrendszerek egyik legfőbb közös jellemzője [14], ami szerint a felhasználói interakciók, események kezelése központilag vezérelt,

ahol minimális beavatkozásra van szükség a fejlesztőtől. A fejlesztőknek elég csak az eset specifikus logikát megvalósítani, azok függőségeit a keretrendszer kezeli.
A keretrendszerek többségének használatakor a keretrendszer birtokolja a vezérlést és komponenseink megfelelő szolgáltatásai úgy érhetők el, hogy a komponenseket valamilyen módon az adott keretrendszer számára regisztráljuk. Ennek egyik lehetősége, ha egy előre definiált interfészt implementálunk az adott komponenssel, amellyel olyan felületet biztosítunk komponenseink felé, amelyeken keresztül a szükséges viselkedés a keretrendszer számára elérhető. EJB3 megjelenése előtt az EJB komponenseknek is előre megadott EJB interfészeket kellett implementálni, melyekkel az „EJB Container”-ek kezelni (például: aktiválni) tudták őket.
A keretrendszer magját alkotó modul, úgynevezett „Inversion of Control Container” tartja kezében a vezérlést, végzi a függőségek kezelését, azaz a referenciák inicializálását, átadását. A „Vezérlés inverziót” a fejlesztési keretrendszerek „Inversion of Control Container”-ek alkalmazásával támogatják úgy, hogy a függőségek kezelése a keretrendszer hatáskörében marad.
A keretrendszerek a függőségek kezelésére általában „Létrehozási tervezési mintákat” használnak („Elvont gyár”, „Gyártó metódus”), amelyekkel a függőségek létrehozásának kezelése optimalizáltan megoldható. [38]
A létrehozott komponenseket a regisztrált függőségek alapján „Dependency Injection” elv alkalmazásával inicializálják a megfelelő helyeken. A „Függőség injekciónak” három változata van:

- „Interface injection”: A létrehozott objektum bekötésére egy interfész és annak egy metódusa alkalmas, amelyet, ha implementál egy osztály, akkor az osztály egyede fogadni tudja az adott objektumot.
- „Setter Injection”: A létrehozott objektumot a cél komponens egy névkonvencióval (az interfészt ezzel helyettesítik) előre az adott objektum bekötésére szánt metódussal köti be.
- „Constructor Injection”: A cél komponensnek nem kell „setter” metódus vagy fogadó interfész, a konstruktorban kerül sor a bekötésre.

A „Sablonfüggvény” tervezési minta jó példa a „Vezérlés inverzióra”, mivel „Sablonfüggvény” esetén a szülőosztály definiálja a keretét a lefutásnak, az algoritmusnak, amelynek az egyes lépései olyan metódusok, melyek interfészét a szülőosztály, megvalósítását és viselkedését az alosztályok adják meg.
A jelenlegi keretrendszerek tekintetében kiemelt fontosságú az „Inversion of Control” és a „Dependency Injection” elvek alkalmazása (JBoss Seam [19], Spring [49]), amelyek már alap eszközrendszerükkel próbálják támogatni az objektumorientált tervezési elvek érvényesülését a megfelelő minőségű megoldások elérése érdekében.

Függőség inverzió elve („Dependency Inversion Principle”): [14][34]
Magasabb szintű modulok ne függjenek alacsonyabb szintű moduloktól, mindig absztrakciótól függjünk! A függőség iránya a konkrét megvalósítások felől az absztrakciók felé mutasson!
Elsődleges cél, hogy a magasszintű komponenseket külön válasszuk az alacsony szintű komponensektől azért, hogy az alacsony szintű komponensek cserélhetőek legyenek. A magas szintű és alacsony szintű komponenseket külön csomagokba

szervezzük. A magas szintű komponensek csomagjában kell definiálni azokat az interfészeket, amelyek a magas szintű szolgáltatások által elvárt viselkedésszolgáltatás felületei. Ezen absztrakt felületeket az alacsony szintű komponensekkel implementáljuk. Látható, hogy a függőség az absztrakció felé irányul, tehát helyes. Ez lehetővé teszi, hogy az alacsony szintű komponensek cserélhetők legyenek, mivel a magasabb szintű komponensek csak az általuk specifikált felületeken keresztül „látják” az alacsony szintű komponenseket.
Észre kell vennünk, hogy a „Vezérlési inverzió” elve és a „Függőség inverzió” elve összefügg. A „Vezérlés inverzió” elve szerint a vezérlést egy központi modul irányítja, amely az alacsony szintű komponenseket – közöttük a fejlesztői komponenseket – előre megadott interfészeken keresztül érheti el. (Lásd az előző bekezdésben „EJB Container” működését!)
Ennek megfelelően a „Vezérlés inverzió” elvénél ismertetett „Dependency Injection” kulcsszereplő a „Függőség inverzió” elv teljesülése számára is, mivel lehetőséget, felületet biztosít az alacsony szintű komponensek futás idejű magas szintű komponensekhez kötésére.
A „Sablonfüggvény” tervezési minta jó példa a „Függőség inverzió” elvére is, mivel a „Sablon függvény” absztrakt szülő osztálya az algoritmushoz egy olyan keretet definiál, amely egyes lépéseivel szemben elvárt viselkedését az azokhoz létrehozott absztrakt metódusokkal specifikálja. Az alosztályok, mint alacsony szintű komponensek adják meg a tényleges működést, betartva az absztrakt metódusokkal deklarált követelményeket.

Nyitottság – zártság elve („The Open-Closed Principle”) [38][48]:
A rendszerkomponenseknek kiterjeszthetőség szempontjából nyitottnak, módosíthatóság szempontjából zártnak kell lennie. Eszerint meg kell kísérelni úgy megtervezni a rendszerünket, hogy ne legyen szükség a változtatásra. A későbbi változások bevezetését kiterjesztéssel lehessen megtenni, amelyhez ne legyen szükség a már meglévő komponensek módosítására. Az elv teljesülése szempontjából a következő két kritériumnak kell megfelelni:

- Kiterjesztésre nyitottság: Az új követelményeknek megfelelő viselkedések alkalmazása a már meglévő modulok kiterjesztésével legyen megvalósítható.
- Módosításra zártság: A meglévő modulok forrását ne módosítsuk. [48]

A kritériumok teljesülése az absztrakció, polimorfizmus, öröklődés és felületek, interfészek megfelelő alkalmazásával oldható meg [48], melyek alkalmazásának módjait az előzőekben definiált tervezési elvek adják meg: „Programozzunk felületre megvalósítás helyett”, „Liskov féle helyettesíthetőség elve – LSP”, „Objektum összetétel használata öröklés helyett, ahol lehet”, „Függőség inverzió elve”.
Azaz a „Nyitottság – Zártság elvének” teljesülése a többi elv egyfajta integrációs elveként is felfogható.
Természetesen az elv összes modulra való teljesülése nem megoldható, de törekedni kell arra, hogy minél több modul teljesítse azt.[48]

Az elv alkalmazása a rendszer integritása szempontjából is nagyon fontos, hiszen, ha egy modul-osztály változik, annak felhasználási helyein kompatibilitási problémák lehetnek. [38]
A „Sablonfüggvény” és a „Stratégia” tervezési minták elsődleges célja az elv teljesülésének biztosítása. [38]

Interfész elkülönítés elve – „Interface Segregation Principle” [34][38]:
A komponenseink interfészeinek meghatározásával kapcsolatosan ad útmutatást, miszerint egy interfész csak olyan metódusokat tartalmazzon, amely ténylegesen szükséges az adott interfész által „vállalt” viselkedés eléréséhez. Ha olyan metódusokat is deklarál, ami nem szükséges az interfész által megvalósítandó feladathoz, akkor ez felesleges fordítási függőséget eredményez.
Egy osztály interfészeit, mint metódus csoportokat kell értelmeznünk, melyek különböző kliensek felé nyújtanak valamilyen, az egyes interfészek által meghatározott szolgáltatásokat. Ezek szerint a kliensek egy része a tagfüggvények egy részét „használja”, melyeket az adott viselkedést szolgáltató interfészen keresztül érnek el, ugyanakkor a kliensek egy másik része a tagfüggvények egy másik halmazát éri el egy másik viselkedést deklaráló interfészen keresztül.
Az osztály interfészeit úgy kell meghatároznunk, hogy azok csak olyan metódusokat specifikáljanak, melyek az általuk behatárolható viselkedés szempontjából lényegesek.

Az osztályok csak egy „dologért” feleljenek – „Single Responsibility Principle” [38]:
Minden osztály esetén csak egy oka legyen, hogy azt meg kelljen változtatni, vagyis az osztályok viselkedését úgy kell behatárolni, hogy csak egy „feladatért” feleljenek, más típusú feladatok ellátására külön osztályokat kelljen bevezetni. Ha programunkat nem így szervezzük, vagyis több feladatot is egy osztállyal akarunk ellátni, akkor, ha valamelyik feladat szerinti viselkedést változtatni kell, kihathat a másik feladat szerinti viselkedésre is, felesleges kockázatot és költségeket generálva.

Általánosan kijelenthető, hogy az előzőekben taglalt alapelvek alkalmazásával az objektumorientált rendszerek implementációs függőségei csökkenthetők. Ezen implementációs függőség csökkentésére adnak segítséget a tervezési minták [15], amelyek az itt bemutatott elvek szerinti megoldáskeresések eredményei.

Ezen objektumorientált alapelvek a dolgozatban bevezetésre kerülő döntésalapú megközelítéssel részben új értelmet kapnak, amelyre az áttekintést követő fejezetekben kitérek.

2.3 Refaktorálás

Az objektumorientált programozás terjedése során sok rosszul megtervezett, kevésbé tapasztalt programozók által megírt rendszer jelent meg, amelyek nem hatékonyak, nehezen karbantarthatók és bővíthetők.
Válaszként olyan technikák születtek, melyekkel a rendszerek felépítése, strukturális integritása javítható. A program minőségét javító transzformációkat refaktorálásnak nevezték el.[13]
A refaktorálás olyan szoftver változtatás, amely nem érinti a kód külső viselkedését, azaz csak a kód belső struktúrájára vonatkozik. Egy program refaktorálásakor a terv-felépítés utólagos javítására van lehetőség. [13]
A refaktorálás érdekessége, hogy nem a tervezés folyamatának minőségét akarja javítani, hanem a már előállt kód minőségének javítása a cél. Emiatt jól kiegészíti a tervezési alapelvekkel és tervezési mintákkal kapcsolatos elveket, amelyek a tervezés folyamatának minőségét tudják javítani.
A már létező refaktorálási módszereknél [18] lényeges észrevétel a feltételes utasítások viselkedésének egy polimorf metódussal történő definiálása, ahol a feltételes utasítás egyes ágait külön-külön alosztályokban megadott felülírt metódusokkal adjuk meg, ahol a szülőosztályban a döntés felületét egy absztrakt metódus adja. Ennek megfelelően az öröklődésekkel létrejövő osztályhierarchiákat, mint döntés absztrakciókat képzelem el, amit a dolgozatban részletezek.

2.4 Objektumorientált programozási metrikák

A mérnöki szakma a létrehozott termékek minőségének jellemzésére mérőszámokat használ. Más mérnöki szakterületekkel ellentétben a szoftverfejlesztésben nincsenek a minőséget egyértelműen meghatározható mutatószámok, csupán indirekt mérőszámokat definiálhatunk, amelyek valamilyen aspektusból teszik mérhetővé az adott szoftver minőségét. [1]
A projektvezetőknek és a tervezőknek az ismert metrika és mérő eszközrendszerek közül nem egyértelmű a megfelelő metrikák kiválasztása, melyekkel a program minőségét a kívánt szempontok alapján mérni tudják.[4] Ennek oka az, hogy a mérhető tulajdonságok száma végtelen, ezért fontos, hogy olyan metrikákat használjunk, amelyeknél egyértelműen bizonyított, hogy az adott metrika értékeit figyelembe véve milyen rendszer tulajdonság, vagy a gyártási folyamat mely vetülete javítható.
Ahhoz, hogy ez egyértelművé váljon, a metrikák céljait meg kell határozni, és a céljaik szerinti osztályozást el kell végezni.

A metrikákat feloszthatjuk folyamat és termék metrikákra [7][35], amely felosztás szerint azok elsődleges céljait a következők szerint összegezhetjük [4][35]:

- Az előállítási folyamat mérhetősége, kontrollálhatósága és javítása.
- A termék minőség, érthetőség biztosítása, javítása – Szoftver tulajdonságok elemzése.

A folyamat és termék metrikák segítik a szervezés támogatásával az ütemezés, költségvetés előállítását, kontrollinghoz mutatószámok biztosítását, illetve a

megvalósítás támogatásával az elemzés, tervezés, fejlesztés, dokumentálás és tesztelési feladatok minőségét javítják. [7]
A tervezés és a rendszerterv a leglényegesebb terület, amely minőségének mérése kritikus eleme a rendszerfejlesztésnek, hiszen a tervezés során a rendszer felépítése már eldől. Kijelenthető, hogy a tervezők a tervezés során nagy szabadságfokkal dönthetnek a rendszer felépítését érintő kérdésekben, ahol a tervezési tapasztalatnak a terv minősége tekintetében kritikus szerepe van, és a tervezői szabadság nagy kockázatot jelent. Kezdő tervezőknél a megfelelő tapasztalat gyakran nem áll rendelkezésre, ezért kiemelt fontosságú különböző tervezési minták (ld. „2.6 Tervezési minták" fejezet), újra használható modulok használata, és a megfelelő minőségbiztosítási lehetőségek – metrikák alkalmazása.
Ezért a terv minőségbiztosítása, azaz a programminőség terv alapján történő vizsgálata kiemelt cél a metrikák alkalmazása szempontjából. [7] Tehát a tervezést támogatni kell heurisztikák alkalmazásával. Ezek közül kiemelt jelentőségűnek tartják az osztályok közötti függésegeket mérő metrikákat, amelyek egy része az osztályok közötti metódushívásokat vizsgálja. Ennek oka – a „2.2 Az objektumorientált tervezés alapelvei" fejezetben leírtakkal összhangban – az, hogy az objektumorientált rendszerek minőségi kérdéskörének kulcspontja a csatoltság és függőség mértéke.

Részben az előbbiekkel összhangban, részben kiegészítve a metrikák céljaként a követelményt, mint elemzés terméket, tervet, programot, illetve annak megvalósításának ellenőrzését, hatékonyságának mérését, illetve a tesztelés hatékonyságának biztosítását jelölték meg. [41]
A következő fő célok határozhatók meg, amelyek igazak a tradicionális és az objektumorientált metrikák esetén is:

- Tervezéshatékonyság: Hatékony volt-e a tervezés?
- Komplexitás: Az architekturális komplexitás csökkentése lehetséges a megoldások hatékonyságának növelésével?
- Érthetőség/Használhatóság
- Újrahasználhatóság/Alkalmazás specifikusság mértéke: A terv alkalmazás specifikus-e?
- Tesztelhetőség/Karbantarthatóság

Mindezt összefoglalva a metrikák igazi értéke a rendszer tesztelhetőséggel, megbízhatósággal, karbantarthatósággal való kapcsolata. [4]
Közelebbről nézve tehát, az egyes metrikák a számítási komplexitást mérik magukba foglalva az algoritmusok hatékonyságát, a gép erőforrás felhasználás hatékonyságát, illetve elvontabb komplexitás faktorokat, amelyek a programok létrehozásának, módosításának, megértésének lehetőségét mérik, illetve azt, hogy a felhasználó mennyire hatékonyan tudja használni a programot.

A termékkel kapcsolatos metrikák tovább oszthatóak külső (publikus) és belső (privát) termék metrikákra, amelyeket a célcsoport különbözteti meg egymástól. [35] A belső termék metrikák a belső fejlesztői csapat számára nyújtanak információkat, amíg a külső termék metrikák a felhasználóknak, megrendelőknek szóló információkat adnak.

Ezen osztályozás szerint további csoportokat határoztak meg: [35]
Külső termék metrikák esetén a következőket kell megemlíteni:

- Termék-megbízhatóság, amely pl. a hibaszámmal mérhető
- Funkcionális metrikák: a termék funkcionális hasznosságát méri.
- Teljesítmény metrikák: a program erőforrás felhasználás mértékének becslése, mint például futási sebesség, felhasznált memória.
- Használhatóság metrikák: méri, hogy a program használata, illetve annak megtanulása mennyire könnyű.
- Vásárlási költség metrikák: a program megvételének és használatának költsége.

Belső termék metrikák:

- Méret metrikák, melyek a programok belső méreteinek jellemzésére használhatóak.
- Komplexitás metrikák (méret metrikákhoz kapcsolódnak).
- Stílus metrikák, melyek a program forráskódjának dokumentációjának írásával kapcsolatosan, azok írásának és dokumentáltságának elemzésére alapulva kerülnek meghatározásra.

A belső és külső metrikák viszonya:
Fontos, hogy a belső termék metrikáknak mindig valamilyen külső metrikával párban kell lennie. Mivel a külső metrikák azok, melyekkel a program minőségét a megrendelők mérhetik, egyértelmű, hogy minden belső metrikának a célja az, hogy valamelyik külső metrikával mért tulajdonság belső vonatkozásait jellemezze.

Folyamat metrikák további csoportosítása a következőképpen tehető meg:

- Költség metrikák, amelyek a projekt költségeit határozzák meg. (Pl.: fejlesztés, karbantartás, a dokumentáció összeállításának költségei mérhetők velük.)
- Erőforrás költségek, amelyek a költségmetrikák egy fajtájaként a humán erőforrás felhasználást mérik, és embernapokban vagy emberhónapokban határozhatók meg.
- Előrehaladás metrikák a fejlesztés során a termék készültségét mérik.
- Folyamat megbízhatatlanság metrikák, amelyek a hibák számát határozzák meg.
- Újrahasználhatóság metrikák a fejlesztés során felhasznált újrahasznált modulok számát/arányát határozzák meg.

Megkülönböztethetünk az objektumorientált módszertant megelőzően a procedurális programozás minőségének mérésére használt metrikákat. [41]
A tradicionális metrikák külön vizsgálják a metodológiát és az adatstruktúrákat, amíg az objektumorientált metrikák a funkcionalitás és az adatok egységes vizsgálatát teszik lehetővé, fókuszba helyezve az azokat egységbezáró objektumokat. Az objektumokat fókuszba helyezve a metrikák vagy a belső objektum struktúrák komplexitását mérik vagy külső komplexitásként az objektumok közötti interakciók minőségét.
Tradicionális metrikákra példák:

- „Cyclomatic Complexity” – „CC”: Metódus komplexitását méri.
- „Lines of Code” – „SIZE”: Metódus méretét méri.
- „Comment Persentage” – „COM”: Kommentezés mértéke: A kommenteket tartalmazó sorok számának aránya az összes, nem üres sorok számához képest.

Vizsgálják, hogy a metrikák hogyan viselkednek objektumorientált program minőség mérése esetén. A tradicionális metrikák egy objektumorientált rendszer estén általában a metódusokat vizsgálják valamilyen szempont szerint. Megállapítható, hogy a procedurális rendszereknél kialakított funkcionális tulajdonságokat mérő metrikák tehát az objektumok metódusain keresztül az objektumorientált rendszerekben is jól használhatók.

Vizsgálható, hogy az objektumorientáltság alapkoncepciójával kapcsolatos minőség mérésére mennyire alkalmasak a metrikák, azaz például milyen kapcsolatban vannak az egyes objektumorientált alapkonstrukciókkal.

A tervezés és a terv minőségbiztosítása a „MOOSE” metrikákkal támogatható [7], amelyek olyan objektumorientált metrikák, amelyek némelyike alkalmas a programfejlesztés életciklusának korai fázisában felismerni a tervezési hibákat.
Az objektumorientált programok minőségének mérésére a „MOOD” objektumorientált metrika gyűjteményt használják, amelyek főbb jellemzői:

- Objektumorientált alapparadigmák, mint egységbezárás, öröklődés, polimorfizmus, metódushívás, mint alap objektumorientált konstrukciók vizsgálatát teszik lehetővé.
- Formális definícióval adottak, amellyel a minősítés szubjektivitása elkerülhető.
- „Méret független” összehasonlítási lehetőséget biztosít a projektek között.
- Nyelv független elemzést biztosít, lehetővé téve eltérő platformon lévő rendszerek összehasonlítását.

Minden „MOOD” metrika az objektumorientált paradigmák alap struktúráira vonatkozik a következők szerint:

- Bezárás (információelrejtés):
    - „Method Hiding Factor – „MHF”: Metódus bezárási faktor
    - „Attribute Hiding Factor” – „AHF”: Attribútum bezárási faktor
- Öröklés
    - „Method Inheritance Factor” – „MIF”: Metódus öröklődési faktor
    - „Attribute Inheritance Factor” – „AIF”: Attribútum öröklődési faktor
- Polimorfizmus:
    - „Polymorphism Factor” – „POF”: Polimorfizmus, többalakúság faktor
- Metódushívás/üzenet küldés:
    - „Coupling Factor” – „COF”: Csatoltság faktor

A belső termék metrikák egy másfajta felosztását ismertetem a következőkben, amelyek a „MOOD” [7][41] metrikákhoz hasonlóan az objektumorientált alap paradigmákhoz kötöttek.
A következő csoportokat határozták meg:

- Méret metrikák
- Csatoltság metrikák
- Kohéziós metrikák
- Öröklési metrikák
- Információ elrejtés-bezárás metrikák
- Polimorfizmus metrikák
- Újrafelhasználásos metrikák

A megadott csoportok jellemzése példa metrikák megadásával: [1][41]

- Méret metrikák („Size Metrics”):
    - „Weighted Methods Per Class” – „WMC” – Az osztályokat a benne deklarált metódusok számával vagy a benne definiált metódusok komplexitását jellemző mérőszámok (pl.: „Cyclomatic Complexity”, amely leírása lentebb található) összegével jellemezzük.
- Csatoltság metrikák („Coupling Metrics”): A csatoltság a program komplexitását növeli csökkentve az egységbezárást, újra felhasználhatóságot, karbantarthatóságot és a kód átláthatóságát.
    - Adatabsztrakciós csatoltság – „Data Abstraction Coupling” – „DAC”: Megadja, hogy egy osztály hány absztrakt adattípusú tagváltozóval rendelkezik. Az absztrakt adattípus egy olyan programozási struktúra, amely képes már meglévő típusokat felhasználva új adattípust létrehozni definiálva az adott típus adatstruktúrájának kezeléséhez szükséges metódusokat, azaz egységben kezeli az adott típus adatstruktúráját és funkcionalitását.
- Kohéziós metrikák („Cohesion Metrics”): Egy modul elemeinek kapcsolódási fokát tárják fel. Egy modul erős kohéziós erővel rendelkezik, ha a rendszer egy olyan szeparált tulajdonságát valósítja meg, melynek egyéb modulokkal nincs vagy kevés kapcsolata van. Hatékony objektumorientált terv maximalizálja a kohéziót, ami összefügg az egységbezárással.
    - Erős osztály kohéziós metrika – „Tight Class Cohesion” – „TCC”: Az adott osztály publikus metódusait vizsgálja páronként úgy, hogy mely pároknak vannak közösen használt tagváltozói. Az így meghatározott közös változót használó metódus pároknak vesszük az összes metóduspárhoz vett arányát, amely megadja az osztály esetén a „TCC” metrika által mért értéket.
- Öröklési metrikák:
    - Öröklési fa mélysége – „Depth of Inheritance Tree” – „DIT”: Egy osztályöröklési fa mélysége a fa hierarchiában vagy gráfban a maximális él szám, amelyen keresztül az adott osztálytól a szülő osztályig el tudunk jutni. Többszörös öröklés estén a maximális él

szám megkötése alapján abba az irányba kell vizsgálni a szülőosztályokat, amerre az él szám, azaz a szülőosztályok száma a bejárás során nagyobb.

- Információ elrejtéses metrikák – „Information Hiding Metrics”: Bezárás, információelrejtés („Informatin Hiding”) az adatabsztrakció alapfogalma, amely teljesülésével kapcsolatos mutatószámok meghatározása természetes igény. Lehetetővé teszi, hogy egy osztály belső működését úgy alakítsuk át, hogy az a belső megvalósítás részeként rejtve marad az általa nyújtandó szolgáltatást publikáló interfész mögött. Mérésére bevezetett metrikákra a következő példák adhatók:
    - Metódus bezárás faktor – „Method Hiding Factor” – „MHF”: A metódusok külső láthatóságának arányát, vagyis az osztály interfészét adó metódusok arányát vizsgálja az összes metódus számához viszonyítva.
- Polimorfikus metrikák („Polymorphism Metrics”): A polimorfizmus, többalakúság lehetővé teszi, hogy egy adott metódus implementációja dinamikusan, futás időben kerüljön meghatározásra – kései kötéssel – a hozzá tartozó objektum típusának megfelelően.
    - Polimorfikus faktor – „Polymorphism Factor” – „PF”: A „PF” metrikával lehetőség van arra, hogy a metódusok felüldefiniálások mértékét meghatározzuk egy adott osztály hierarchiában.
- Újrafelhasználási metrikák: Az objektumorientált programozás létrejöttének fő oka a kód és terv újrafelhasználásának elősegítése volt. Ezért fontos az újrafelhasználás mértékének mérése, amire a következő metrikák adnak példát:
    - Újrafelhasználási ráta – „Reuse Ratio” – „U”: Szülőosztályok számának az összes osztály számosságához viszonyított aránya.

Érdemes megemlíteni a kivételkezeléssel kapcsolatos metrikákat [2], mivel a megfelelő kivételkezelés alkalmazása kritikus része a rendszereknek. A következő metrikák említhetők:

- Kivételkezelő blokkok száma egy osztályban metrika – „Number of Catch Blocks Per Class” – „NCBC”
- Kivételkezelés faktor – „Exception Handling Factor” – „EHF”: Azt szemlélteti, hogy a kivételkezelés osztályainak száma hogyan aránylik a kivétel kezelési blokkokhoz.

A ciklikus komplexitás metrika – „Cyclomatic complexity” – „CC” [16] a metódusok vizsgálatát a bennük definiált algoritmusok elemzésével végzi, amely során a benne lévő elágazások számát elemzi. A programokat gráfként vizsgálja, ahol a csomópontok az elágazást végrehajtó utasítások, az élek pedig az elágazások közötti átmenetek.

$$CC = \text{Élek száma} - \text{Csomópontok száma} + 2$$

Nagyobb komplexitást jelent a nagyobb döntés szám az adott metóduson belül, ami nem javasolt.

A „Quality Indicator” – „Qi” [4] metrika használatával különböző szoftver attribútumok, mint például komplexitás és együttműködés/függőség, mint osztályok közötti kapcsolatok együttesen vizsgálhatóak a vezérlési útvonalak és azok valószínűségeinek felhasználásával. Az új metrika megnevezése: „Quality Indicator” – „Qi”. Ezzel több már meglévő objektumorientált metrika által nyújtott ellenőrzés integrálható. Az alkotók kiemelt célja, hogy a szoftver tesztelhetőség és karbantarthatóság külső szoftver tulajdonságra gyakorolt hatását igazolják. Céljai bővebben kifejtve:

- A magas kockázatú osztályok azonosítása, amelyek esetén a minőségbiztosítás magas tesztelési költséggel jár.
- Hibára hajlamos osztályok azonosítása.
- Egy osztály módosításának a rendszer többi részére gyakorolt hatásának mérése.

A „Qi” a vezérlési hívási gráfon („Control Call Graph”) alapuló metrika, amelynél figyelembe veszik az osztályok együttműködését is. A metrika értékek meghatározásánál a metódushívások valószínűségét is felhasználják. Az eredmény egy normalizált érték, amely 0-1 közé esik.
Vezérlési hívási gráf („Control Call Graph – CCG”):
A „CCG” a „CFG” – „Control Flow Graph”-nak egy egyszerűsített változata, ahol azon csomópontok, amelyek nem metódushívások, ignorálva vannak.
Érdekesség a polimorfikus hívások gráffal való ábrázolásának megjelenése, ahol a dinamikus kötések, azaz metódushívások lehetőségeit gráf élek szemléltetik elágazásokkal mutatva a polimorfikus lehetőségeket.
A vezérlési hívási gráf összeállítása statikus kódelemzéssel lehetséges.
Qi meghatározása:

- Statikus kód analízissel a vezérlési hívási gráf összeállítása.
- Vezérlési hívási gráf alapján „Qi” értékek meghatározása minden metódusra. A „Qi” érték meghatározása az egyes metódusokra azok belső paraméterei alapján történik, amelyeken azok „ciklikus” komplexitását („cyclomatic complexity”), unit tesztjeinek kiterjedését illetve az általa hívott metódusok „Qi” értékeit értjük. Ez azt jelenti, hogy ha egy metódus minősége nem megfelelő, mert például a unit tesztekkel történő lefedettsége nem megfelelő, akkor annak hatása van az összes az adott metódust hívó metódusra is.
- Osztály „Qi” értékek meghatározása.

Gráf él valószínűségek bevezetése:
A metódusok vezérlési hívási gráfja (CCG) felfogható, mint lefutások halmaza, amelyeken a vezérlés végig futhat. A vezérlési struktúrákban lévő feltételektől függenek a lefutási útvonalak. A vezérlési folyamatok valószínűségének meghatározásához a vezérlési hívási gráf minden lefutásához rendelünk egy valószínűséget. Ezt úgy határozzuk meg, hogy a vezérlési útvonal minden egyes éléhez rendelünk egy valószínűséget, amelyeket összegzünk, és megkapjuk az egyes útvonalak valószínűségét.

$P(C_k) = \prod_{i=0}^{n_k} P(A_i)$, ahol az A_i a C_k lefutás élei.

A valószínűségek meghatározásánál egy statikus megoldást használnak, amellyel az egyes feltételes vezérlési szerkezetek egyes lehetőségeinek valószínűségét egyformán határozzák meg. A dinamikus valószínűség meghatározás lehetőségét nyitva hagyják.

A „Qi" meghatározása egy metódusra a következő módon történik:

- Meg kell határozni a metódus belső minőségi mutatóját (Instrintic Quality Indicator), amely a tesztelhetőségével kapcsolatos mutatószám.
- Az adott metódus által hívott metódusok „Qi" értékét összegezni kell, mindegyiket súlyozva az adott metódus vezérlési szerkezetnek megfelelő hívási valószínűségével.

A M_i metódus számára a „Qi" metrika (Qi_{M_i}) megadása a következő módon lehetséges:

$$Qi_{M_i} = Qi^*_{M_i} * \sum_{j=1}^{n_i} \left(P(C_j^i) * \prod_{k \in \sigma} Qi_{M_k} \right)$$, ahol

Qi_{M_i}: Az M_i metódus Qi értéke.

$Qi^*_{M_i}$: Az M_i metódus belső Qi értéke.

$P(C_j^i)$: Az M_i metódus C_j^i lefutásának valószínűsége.

Qi_{M_k}: A C_j^i lefutás metódusainak Qi értékei.

n_i : Az M_i metódus CCG szerinti lefutásainak száma.

$Card(\sigma) = m_j$: A C_j^i lefutás metódusainak száma.

A dolgozatomban ismertetett döntés struktúra minőség, döntésredundancia elemzés elveit felhasználva olyan új metrikákat vezetek be, amelyek a vezérlési folyamat gráfok minőségének kérdését feszegetik, amely javítására a dolgozatban ismertetett módszer ad lehetőséget.

2.5 A program formális leírásának módszerei

A programformalizálás módszereinek célja a program szerkezeti formalizálásán túl a program viselkedésével, azaz folyamataival szemben elvárt követelmények specifikálása, amelyre alapozva program minőséggel kapcsolatos elemzésekre nyílik lehetőség.

A Floyd féle induktív állítások módszere [12] és Hoare féle deduktív levezetési rendszer [17] szerint a program helyessége a parciális és totális helyesség bizonyításával igazolható, ahol a program viselkedését elő és utófeltételek deklarálásával adjuk meg.

A program viselkedésének specifikálására elő- és utófeltételek alkalmazásával jól kezelhető megoldás adható, amellyel elemi program elemektől kezdve

programstruktúrákig lehetőség van viselkedéssel kapcsolatos követelményrendszer deklarálására.
A Hoare féle formális ellenőrzés alapjain alakult ki a „Design by contract”- DBC programfejlesztési módszertan, amely szerint egy osztály és annak felhasználója között egy „szerződés” van, melynek megfelelően a felhasználónak deklarált feltételeket kell teljesítenie az adott osztály metódusának hívásakor. A metódus visszatérésekor az osztály is garantál bizonyos feltételeket, amelyek a metódus elvárt működését deklarálják. [28]
Egy program/objektum viselkedése egy un. viselkedési szerződéssel azonosítható [28][44], amelyben a szerződés a program és objektumok lehetséges viselkedésének halmazát jelenti. [44] („Concept of a behavioral contract” – „Design by Contract”)
Ezen szerződéseket elő- és utófeltételekként specifikálhatjuk.
A logikai formalizmusok rendszerét egészíti ki a programok viselkedésének leírása szempontjából kifejező „Temporális logika” [24], ahol fontos elemként jelennek meg az állapotok, mint a program viselkedésének fő mutatói. Az állapotok halmazára a program futása során kifejező megkötéseket tehetünk deklarálva az állapotátmenetek lehetőségeit, amelyek a program működésének megértését növelhetik.
Ezen formalizációs módszerekre épülve jelent meg a JML – Java Modelling Language [8][9][25][26][27] Java alapú viselkedés interfész specifikációs nyelv, mellyel a Java programok szintaktikai interfészét és viselkedését lehet specifikálni [8][25]. Szintaktikai interfészen a Java interfészek és osztályok programozói interfészét, azaz a metódusaik fejlécét, illetve változóik nevét és típusát értjük. Megfelelő JML bejegyzésekkel ezen interfészek viselkedését is specifikálni tudjuk. Megfogalmazható tehát, hogy hogyan és mire kell/lehet használni az adott osztályt, metódust. [8] Az interfészek egy modulra vonatkozóan mutatják be a hozzáférést a felhasználónak, illetve a viselkedésüket bemutatva specifikálják, hogy mit nyújt a felhasználónak [25].
A JML specifikációs nyelv az Eiffel típusú szintaktikát keveri a VDM és Larch modell alapú szemantikájával úgy, hogy az Eiffelben használatos feltételeket Java kifejezésekkel teszi könnyebben használhatóvá, a VDM-ből és az Larchból ismert eszközökkel téve azt kifejezőbbé. [25][27]
Ki lehet jelentenünk, hogy – az említetteken túl – számos állapot alapú specifikációs nyelv alap specifikációs eszközei használatosak a JML-be építve, például az elő- és utófeltételek, állítások, invariánsok.[9] De ezen eszközök önmagukban még nem alkalmasak arra, hogy megoldást adjanak az objektumorientált nyelven írt programok formális alapú moduláris ellenőrzésére. Erre a JML a keretállítások, adatcsoportok, szellem és modellváltozók használatának bevezetésével próbál megoldást találni [9], amely lehetőséget ad arra, hogy egy osztály követelményeit absztrakt szinten deklaráljuk, amelyek implementációs „kötését” azon a származtatási szinten adjuk meg, ahol az ténylegesen megvalósul. Ezzel az eszközzel tehát egy osztály követelményének a konkrét implementációs megkötésektől mentesített deklarációja lehetséges, amely rugalmasan teszi lehetővé a modulok viselkedésének bemutatását.
A JML specifikációkat lehet megjegyzésként a forrás állományokba is írni (amit a Java fordító nem fog értelmezni, mivel megjegyzéseknek tartja őket) vagy

elkülönített specifikációs állományokba. A specifikációs állományok specifikációi öröklődési hierarchiába szervezhetők, jelentősen megkönnyítve jól strukturált specifikáció létrehozását.
Lehetséges, hogy az adott forrásállományt teljes körűen specifikáljuk, amit „Heavyweight” specifikációnak nevezünk, illetve lehetőség van csak részlegesen specifikálni az adott forrásállományt, amit „Lightweight” specifikációnak tekintünk.

A JML-t alkalmasnak tartom arra, hogy a programok viselkedésének specifikációját kifejezően modellezze. Ezért a JML-re alapozva fogom a dolgozatban bemutatásra kerülő döntésalapú program szervezési megközelítést vizsgálni, továbbá a döntésalapú megközelítés szerint vizsgált tervezési minták formalizálásának lehetőségeit JML-re alapozva elemzem. [31]

2.6 Tervezési minták

A tervezés minőségét jelentősen javíthatja, ha a tervezők a tervezés alatt olyan „recepteket” kaphatnak, melyekkel a korábban használt és már bevált tervezési megoldások újra használhatók. A minőségi programok előállítását olyan szabályok garantálhatják, melyek a tervezési tapasztalatok alapján kerülnek megfogalmazásra.
Célszerű tehát a tervezés szempontjából korábban bevált megoldásokat szabályokként megfogalmazni és azok újrahasznosítását elősegíteni. Ezért különböző objektumorientált rendszerekben hasonló osztály és objektumszerkezetű struktúrákat találhatunk, amelyekkel az újra és újra feltűnő hasonló tervezési problémákra adnak a szakemberek hasonló megoldásokat. Ezeket a visszatérő tervezési problémákra adott újrahasznosítható tervezési megoldásokat, általános érvényű „recepteket” hívjuk tervezési mintáknak, amelyek a minőségi terv és megvalósítás eszközei. Specifikusabban egymással együttműködő objektumok és osztályok leírásai, amelyek esetre szabottan valamilyen általános tervezési problémát oldanak meg.[15]
A tervezési minták a tervezési tapasztalatok összegyűjtésének eredményeként jöttek létre, melyek a tervezés során a tervezőknek tanácsot adnak, hogy miként tudják az adott tervezési helyzeteket megfelelően megoldani. Számos esetben hasonló, általánosítható tervezési problémával szembesülünk, amelyre érdemes szakmailag megvitatott és elfogadott válaszokat megfogalmazni és azokat esetspecifikusan felhasználni.

A tervezési minták céljai: [15]

1. Osztályok, objektumok meghatározásának támogatása: Mint ahogy az alapelvek ismertetésénél részleteztem, komplex feladat a problémakör szerinti objektumok meghatározása, azok osztályokba szervezésével a megfelelő absztrakciós szint biztosítása. A tervezési minták ezt a folyamatot támogatják tipizált eseteket biztosítva az egyes absztrakciós szituációk ábrázolásához.
2. Objektum felületek, közöttük lévő kapcsolatok meghatározásának támogatása: Objektum felületen értjük az adott objektumnak küldhető üzenetek halmazát, más terminológia szerint azon metódusok halmazát,

amelyen keresztül az adott objektumhoz hozzáfér a környezete. A tervezési minták meghatározzák, hogy az adott esetekben a részvevő objektumoknak milyen felületet biztosítsunk. Lényeges szerepet játszik a tervezési minták által javasolt megoldások során a polimorf működés elősegítése, amely a megfelelő dinamizmust biztosítja. Például a „Stratégia” tervezési minta megszabja, hogy az egyes „Stratégia” megvalósításoknak milyen közös felületének kell lennie ahhoz, hogy azok dinamikus használata, bekötése megoldható legyen. Más tervezési minták további megszorításokat irányoznak elő az objektumok felületével kapcsolatban, mint például a „Látogató” tervezési minta esetén a „Látogató” osztály felületének minden látogatott osztályra vonatkozó metódussal kell rendelkeznie, amelyet névkonvenciók betartásával kell biztosítani.

3. Felületöröklések használatának biztosítása a megfelelő absztrakciós lehetőségek kihasználásával. Például az „Összetétel” közös felületet ad az elemi és az összetett alosztályok objektumai számára.
4. A megvalósítások absztrakt felületek mögé rejtésével az implementációs függőségek megszüntetésének elősegítése. Természetesen a példányosítást el kell végeznünk valahol, azaz meg kell adni a tényleges megvalósítást, ami után már csak egy absztrakt felületen keresztül érjük el az objektumot. A létrehozási minták a megvalósítás megadásának módjára adnak sablonokat, amelyekre szintén absztrakt felületeket biztosítanak, azaz a létrehozáskor is csupán már egy absztrakt felületen keresztül érhető el a létrejövő objektum, annak tényleges megvalósítása rejtve marad. A függőségek csökkentésével elősegíthető az osztályok, objektumok, komponensek laza csatolása, amely kritikus a karbantarthatóság, újrahasználhatóság tekintetében.
5. Segítenek eldönteni, hogy az osztály öröklés vagy objektum összetétel használata előnyösebb egy adott esetben. A cél, hogy a két eszközt felhasználva egy harmonikus osztály és objektum szerkezet keletkezzen. Ezzel kapcsolatos dilemma feloldásában segítenek a tervezési minták, amelyek javaslatokat tesznek az összetételek és öröklések megfelelő arányú használatára.
6. Objektum összetétel és képviselet/aggregáció, mint az objektum összetétel felhasználás egy formájának, sablonizált felhasználásának elősegítése, amivel az objektum összetételek érthetetlen szövevénye ismert szerkezetű sablonokkal váltható ki. A képviselet/aggregáció tervezési mintával való támogatására példa a „Látogató”, amely esetén egy objektum szerkezeten végrehajtott műveletért egy úgynevezett „Látogató” objektum felelős, amely a szükséges viselkedést dinamikusan szállítja a cél objektumokhoz kiemelve azt a cél objektumszerkezetből úgy, hogy a cél objektumot megkapja a „Látogató”, hogy azon a szükséges műveleteket el tudja végezni.
7. A rendszer futás idejű működésének megértésének elősegítése: A rendszer viselkedése a rendszer statikus tulajdonságai alapján komplex osztály és objektum szerkezetek, azaz összetett problémák esetén nehézkes, amelyben nagy segítséget nyújthatnak a tervezési minták, azon belül is a viselkedési

tervezési minták, amelyek azonosításával az adott programrészek felépítésének indítékait érthetjük meg.

8. Karbantarthatóság, újra használhatóság növelése: Mivel a tervezési minták célja a fentebb említett algoritmikus, környezet függőségek összességében implementációs függőségek csökkentése, az absztrakt felületek meghatározásának elősegítése, ennek megfelelően a karbantarthatóság, a későbbi változások megvalósíthatósága javul.

Ahogy az objektumorientált alapelvek taglalásakor már hangsúlyoztam, látni kell, hogy az objektum összetételek és öröklések használatával kapcsolatosan kritikus annak meghatározása, hogy mikor melyik konstrukció használata optimálisabb az újra használhatóság, karbantarthatóság szempontjából. A cél, hogy a két eszköz felhasználásával egy harmonikus osztály és objektum szerkezet keletkezzen.

A szakmailag elfogadott tervezési minták legjelentősebb gyűjteményét a „négyek bandájának" könyvében (Programtervezési minták: Újrahasznosítható elemek objektumközpontú programokhoz [15]) találjuk, ahol a tervezési mintákat névvel, magyarázattal és értékeléssel ellátva rendszerezik.
A tervezési minták ismertetése során a minták nevével hivatkoznak egy tervezési alapproblémára, az arra adott megoldást illetve következményeket ismertetve.
A tervezési mintákat létrehozási, szerkezeti és viselkedési mintákra bonthatjuk. Mindhárom csoporton belül megkülönböztetünk osztály és objektummintákat. Osztályminták esetén statikus, öröklődés alapú kapcsolatokon alapuló mintákkal foglalkozunk. Objektumminták esetén viszont olyan objektum összetételeken alapuló mintákról van szó, melyek futásidőben változhatnak, azaz dinamikusak. Természetesen éles határok nincsenek, az objektum minták is tartalmaznak öröklődést, de a hangsúly az öröklődés mellett elsősorban az objektum összetételeken van.

A létrehozási minták a példányosítási folyamat elvont ábrázolásai, melyek objektumok, objektum-szerkezetek létrehozásának támogatására adnak megoldásokat, azaz a létrehozási folyamat absztrakciói. Céljuk a rendszer függetlenítése a létrehozási folyamattól. A termék objektumokról a felhasználási helyeken csak azt tudjuk, hogy szolgáltatásai milyen felületen keresztül érhetők el. Létrehozásuk zárt, azaz a termék létrehozási folyamata rejtett. Létrehozási osztályminták a létrehozás feladatát alosztályokra, amíg a létrehozási objektum minták a létrehozás feladatát objektum összetételek alapján támogatják.
Például az „*Építő*" tervezési minta jól használható komplex objektum szerkezetek létrehozására úgy, hogy az objektum szerkezet létrehozása függetlenné tehető az összetevők létrehozásának módjától. *„Elvont gyár"* tervezési minta egymással összefüggő objektumok létrehozását támogatja, az összefüggő objektumokat külön-külön osztályhierarchiába rendezve közös létrehozási felületet biztosítva nekik, melyek már létrehozáskor együtt kezeltek.

Szerkezeti minták öröklést és objektum összetételeket használnak komplex objektumszerkezetek felépítésére. Céljuk ennek megfelelően összetett objektum

szerkezet felépítési lehetőségének megadása. A szerkezeti osztályminták öröklési kapcsolatok kiépítésével hoznak létre különböző osztályok számára közös interfészeket, amíg a szerkezeti objektumminták objektum összetételekkel hoznak létre új szolgáltatásokat nyújtó komponenseket. Az objektum összetételekkel létrejövő szerkezetek dinamikus lehetőségei rugalmasabb megoldásokat adhatnak, mint a szerkezeti osztályminták statikus megkötései.
Az „*Összetétel*" szerkezeti objektum minta alkalmas komplex objektum összetételen alapuló objektumszerkezetek felépítésére, amit a „*Díszítő*" tervezési minta azzal egészít ki, hogy komplex objektumszerkezetek létrehozásakor objektumok szolgáltatásait egészíti ki új lehetőségekkel.

A viselkedési osztály- és objektumminták algoritmusok és működések leírását támogatják. Nem csak az osztály és objektumkapcsolatokat szemléltetik, hanem a közöttük folyó kommunikációt is bemutatják a dinamikus modellezés eszközeit (szekvencia diagramok [40], állapot diagramok [40]) felhasználva.
Az osztályminták az öröklést használják az algoritmusok, működési stratégiák leírásához, amíg a viselkedési objektumminták azt mutatják be, hogy az objektum összetételek miként tudnak valamilyen komplexebb tevékenységet nyújtani. Osztálymintára példa a „*Sablonfüggvény*" tervezési minta, ahol egy absztrakt osztályt/interfészt használhatunk fel, mint keretet egy algoritmus kiépítésekor, amit tartalommal az adott absztrakt felület alosztályaiban megadott konkrét megvalósítás ruház fel. A „*Megfigyelő*" objektum viselkedési minta azonban objektum összetételeken keresztül hoz létre „egy-sok" összerendeléseket, amelyeket felhasználva, ha egy objektum állapota megváltozik, arról minden tőle függő objektum értesül. Ez a minta nagyon jól alkalmazható felhasználói interfészek megvalósításakor. Objektum viselkedési minták esetén tehát az elsődleges szempont a függőségek kezelése, azaz milyen módon valósulhat meg az objektumok közötti kapcsolatok rugalmasságát biztosító laza csatolás.
A viselkedési minták egy csoportja valamilyen adott osztályra vetíthető tulajdonság változatainak egységbe zárását végzi, legyen az akár állapotra, viselkedésre, stratégiára vonatkozó program tulajdonság. Az egyes változatokat egy osztályhierarchiába szervezi egy elvont felülettel egységes hozzáférést biztosítva nekik, amely egyben a felhasználási helyeken a változatok bezárását is biztosítják. Más viselkedési minták esetén az előzőekben említett függőségek – laza csatolás megvalósításával kapcsolatos lehetőségek modellezése a cél.

Lényeges kiemelni, hogy a tervezési minták elsődleges célja a rendszerekben lévő különböző típusú függőségek feloldása, amelyre a különböző típusú tervezési minták eltérő esetekben nyújtanak megoldásokat.
Ezt kiegészítve a tervezési minták célja a kód redundanciák megszüntetése illetve a végrehajtáskor az adatredundanciák feloldása.[20] (Az elképzelés a tervezési minták előzőleg megfogalmazott általános célkitűzését pontosítja.)

Hiánypótló azon elképzelés, amely a tervezési mintákat, mint refaktorálási eszközöket mutatja be, melyek lehetőséget adnak a már létező kódok szerkezetének javítására.[18] A tervezési minták alapján alacsony szintű tervezési

transzformációk, azaz refaktorálási lépések sorozatával tehető meg a terv javítása. Számos olyan szituációt fogalmaznak meg, ahol valamelyik tervezési minta alkalmazásával lehetőség van a kód és terv minőség refaktorálással történő javítására.[18] Ez nagyon lényeges megközelítése a tervezési mintáknak, mivel ezen eset leírások lehetőséget adnak arra, hogy a tervezési minták alkalmazásával javíthassunk a program terv minőségén. Lényeges újítás, hogy a tervezési minták használatát program strukturális helyzetek alapján határozza meg, amely alapján a tervezési minták felhasználásának lehetőségeinek felfedezésére is mód van.

A már ismert refaktorálási megoldások szerint a feltételes utasítások viselkedésének polimorf metódusokkal történő definiálásának lehetőségét figyelembe véve [18], illetve „3. Az objektumorientált programozás döntésalapú vizsgálata" fejezetben megadott döntés, döntéskiemelés, döntésösszevonás koncepciójára alapozva [30] kijelenthető, hogy a tervezési minták célja különböző döntésszerkezetek szerinti döntésredundanciák, mint egyfajta implementációs függőségek feloldása.
Az elképzelésnek megfelelően a tervezési minták tulajdonságainak vizsgálata szempontjából természetesebb osztályozásra is lehetőség van, amit a dolgozatomban ismertetek.

2.6.1 Tervezési minták formalizálása

A tervezési minták lényegének megértését szolgálják a tervezési minták formalizálásának módozatai, melyek különböző szempontok szerint próbálják szemléltetni a tervezési mintákat lényegét, esetenként grafikai ábrázolással egészítve ki azt. (Pl. LePus [10], PDL[3])
A tervezési minták leírása megtehető természetes nyelven, példákkal, OMT diagramokkal [42] szemléltetve azokat, a következő struktúra szerint adva meg azok leírását: [15]

- Minta megnevezése.
- Cél – Rövid leírással a tervezési minta céljának bemutatása.
- Egyéb nevek – A minta egyéb ismert megnevezései.
- Feladat – Tervezési probléma megfogalmazása és megoldásának forgatókönyve.
- Alkalmazhatóság – Milyen helyzetekben alkalmazható a minta?
- Szerkezet – Grafikus szemléltetés OMT diagramokkal.
- Résztvevők – Osztályok és objektumok, amelyek részt vesznek a mintákban.
- Együttműködés – Hogyan működnek együtt a résztvevők?
- Következmények – Minta használatának céljai, előnyei és hátrányai.
- Megvalósítás – Megvalósítás módszere.
- Példakód – Forráskód részletek a megértést könnyítendő.
- Ismert felhasználások – Valós rendszerekből vett példák.
- Kapcsolódó minták – Milyen kapcsolatban állnak egyéb tervezési mintákkal?

Azonban a természetes nyelvvel és példákkal leírt tervezési minták nem pontosak [10][36], ezért a szakma számos megoldást javasolt, hogy a tervezési minták leírása formalizáltan, pontosabban történhessen (LePUS [10], DisCo [36], BPSL[47]).

A tervezési minta formalizálás „evolúció” kezdetén elsősorban olyan megoldások születtek, amelyek a tervezési mintáknak vagy a statikus-strukturális vagy a dinamikus-viselkedés oldalát formalizálták, de nem adtak megoldást mindkettőre.[47] A tervezési minták formalizálását a statikus jellemzők formalizálásával oldotta meg a LePUS [10], amely a szerkezeti jellemzőket logikai formulák segítségével fogalmazta meg. Temporális logikára [24] alapozva jött létre a DisCo [36] formalizációs megoldás, amely a tervezési minták viselkedésének formalizálására tett kísérletet. A tervezési minták statikus, strukturális jellemzőinek és funkcionalitásának a formalizálását is megoldotta a BPSL [47] formalizáló nyelv, pótolva ezzel a korábbi formalizálások hiányosságát. A BPSL [47] a tervezési minták szerkezeti tulajdonságait elsőrendű logika segítségével formalizálja, a minták metodológiáját pedig „Temporális logikára” [24] alapozza.
A tervezési minták tulajdonságainak egzakt ábrázolásán és a megértésen túl a formalizálás célja, hogy az előálló formulákon elemzéseket tudjunk végezni, hogy így a megfelelő következtetések levonhatók legyenek. Ennek érdekében a tervezési minták statikus tulajdonságaira alapozva kidolgoztuk a Simp Language [32] formalizációs nyelvet, mellyel lehetővé vált a tervezési minták, illetve tervezési komponensek formalizálásával előálló logikai formuláknak a nyelv részeként megadott öröklődési szabállyal való ellenőrzésének, logikai vizsgálatának lehetősége konstruktív bizonyítással. A tervezési komponensek és azok részhalmazát alkotó tervezési minták formalizációja Simp Language[32] szerint bonyolult, ezért azt egy leképezési technikával tettük könnyebbé, automatizálhatóvá, amely a tervezési komponensek és tervezési minták OMT diagramjait [42] tudja leképezni Simp Language formulákra.[33] A Simp Language [32] a tervezési minták statikus tulajdonságainak formalizálására alkalmas, ezért kutatásunk arra irányult, hogy miként lehet a tervezés és azon belül a tervezési minták dinamikus tulajdonságait elemezni.
Ahhoz, hogy a tervezési komponensek és a tervezési minták dinamikus aspektusának elemzését elvégezhessük, illetve, hogy a tervezési minták tulajdonságai megfelelő módon ábrázolhatók legyenek, a jelenleg elfogadott értelmezését ki kell terjeszteni.
Az objektumorientált szerkezeteket a döntés struktúrák alapján vizsgálom, amely szerint az objektumorientált módszertan [42] alapeszközeit, közöttük az öröklés/származtatás értelmezését és célját is kiterjesztem. [30] Az elképzelés szerint a tervezési minták célja különböző döntés struktúrák esetén olyan osztály és objektum szerkezetek bemutatása, melyek alkalmazásával a döntésredundanciák feloldhatók.
Az elképzelés alapján JML-re [8][9][25][26][27] alapozva bemutattam [31], hogy az egyszerű és összetett döntésszerkezetek JML alapú formalizációja miként tehető meg. Ezt a megoldást finomítva és kiterjesztve dolgozatomban bemutatom, hogy a tervezési minták döntés szerkezet szerinti JML alapú formalizációja miként végezhető el.

3 Az objektumorientált programozás döntésalapú vizsgálata

A „2. Áttekintés” fejezetben ismertetett objektumorientált tervezési elvek alapján átfogó képet kaphattunk arról, hogy az objektumorientált tervezés során milyen elveket követve juthatunk jól strukturált és felépített rendszerhez („2.2 Az objektumorientált tervezés alapelvei”), bemutattam az „építkezés” eszközeit („2.1 Az objektumorientált programozás alapeszközei”), azok kapcsolódási lehetőségeit, melyekkel a különböző osztály és objektumszerkezetek kialakítása lehetséges. Megvizsgáltuk, hogy az elvek teljesülésének elérésére a tervezési minták hogyan használhatók („2.6 Tervezési minták”). Bemutattam a megvalósítás és végtermék minőségének mérésére milyen metrikák léteznek, azok hogyan használhatók.(„2.4 Objektumorientált programozási metrikák”) A refaktorálás módszerének bevezetésével ismertettem annak lehetőségét, mellyel a kódminőség, kód szerkezet előre megadott szabályok alkalmazásával utólag hogyan javítható. („2.3 Refaktorálás”)
Fontos megjegyezni, hogy számos helyen nem kapunk egyértelmű választ az objektumorientált tervezési elvek alkalmazásának mozgató rugóira, illetve, hogy az egyes esetekben az objektumorientált eszközrendszer mely elemeit érdemes felhasználni. Erre a problémára válaszkeresésként megszületett tervezési minták tényleges strukturális céljai néhol nem egyértelműek.

A refaktorálási technikaként ismert feltételes utasítások viselkedésének osztály-alosztály szerkezetekkel és polimorf metódussal történő megvalósítására alapozva [18] az objektumorientált programozás alapkonstrukcióját, mint döntés absztrakciós technikát értelmezve [30] az objektumorientált alapfogalmakra, tervezési alapelvekre kiegészítő értelmezést adok. Az elképzelés szerint a program minőség mérés, javítás új szempontok szerint bővülhet.
Az új elképzelésre alapozva a tervezési minták céljait is újraértelmezhetjük, miszerint a tervezési minták célja a jól strukturáltság elérése a döntésismétlődések megszüntetésével. Tehát a tervezési minták a döntéskiemelésekre adnak recepteket, vagyis arra, hogy a különböző strukturált döntési helyzetekben a döntésismétlődés megszüntetése miként biztosítható. [30]
Hogy az elképzelésnek megfelelően a strukturáltságot, azaz a jól strukturált program irányelveinek teljesülését elemezni tudjuk, szükséges egy olyan formalizációs eszköz, amellyel a döntés definíciók vizsgálhatók. Erre a Java programok viselkedés interfész specifikációs nyelvét, a JML-t [26] használom [31], amely lehetőséget ad a döntés lehetőségeinek adatstruktúráját, metodológiáját logikai kifejezésekre alapozva specifikálni. Az így formalizált döntések vizsgálata a jól strukturáltság szabályainak való megfelelés szempontjából már lehetséges.

A fejezetben taglalt témakörök szerint új eredménynek a következőket tartom:
Döntés, döntéskiemelés fogalmainak bevezetése
Bevezetem a megvalósítástól független döntés fogalmát, amely kifejti, hogy az öröklődéses osztályszerkezetek a döntések absztraktabb deklarációs módjai, illetve kijelenti, hogy a döntés kiemelés során a döntések viselkedése és adatszerkezete

nem változik. Ezek a definíciók a refaktorálás témakörében már alkalmazott feltételes utasítások öröklődéssel-polimorf metódussal történő definiálásra épülnek. [18]
(3.1 Döntés és döntéskiemelés)

Döntésredundanciák megszüntetése döntésösszevonásokkal
A döntés, döntéskiemelés fogalmára alapozva meghatározom a döntésredundanciák azon eseteit, amelyek elkerülése érdekében a döntésösszevonások végrehajtása indokolt lehet.
Ezek az esetek a későbbiek folyamán kritikusak, mivel ezek motiválják a tervezési alapelvekkel kapcsolatos javaslatok megszületését, illetve a kód minőség mérés lehetőségeit kiterjesztő metrikák bevezetését. A tervezési minták osztályozása szintén ezen esetekre épül.
(3.2 A döntésredundanciák elkerülésének esetei, 3.3 Döntésösszevonás)

Új objektumorientált tervezési alapelvek bevezetése
Látnunk kell, hogy a döntésösszevonás esetei a „2.2 Az objektumorientált tervezés alapelvei" fejezetben taglalt számos elvet új megvilágításba helyeznek. A leglényegesebb, hogy a döntésösszevonás esetei egyértelmű segítséget adnak annak eldöntésében, hogy mikor szükséges és indokolt öröklődés, azaz osztályhierarchia bevezetése.
Bevezetek két olyan új objektumorientált tervezési alapelvet, melyek a döntésredundancia elkerülés szabályaiból táplálkozva fogalmaznak meg két fontos szabályt. Az „Öröklést döntésredundancia megszüntetésre alkalmazzunk" javasolt alapelv azon eseteket deklarálja, amikor öröklődés használata indokolt.
„A döntésredundanciát kerüljük" tervezési alapelv általánosan fogalmazza meg a döntésredundanciák elkerülésének szükségességét, amely a programminőség javulását eredményezi.
(3.6 Objektumorientált tervezési alapelvek kiegészítése)

Döntésredundanciák mérése új metrikákkal
Olyan új objektumorientált metrikákat vezetek be, melyekkel lehetőség van a programban szereplő döntésredundanciák mértékének megállapítására. Ezek közül a legegyszerűbben mérhető a „Döntésabsztrakciós metrika", amely a kiemelt – nem kiemelt (polimorf-nem polimorf) döntésesetek számosságának arányát vizsgálja. A kiemelt döntésesetek arányának növekedése javítja a kód stabilitását.
A „Döntéskiemeléssel létrejött öröklődések aránya" metrikát az „Öröklést döntésredundanciák megszüntetésre alkalmazzunk" új tervezési alapelv teljesülésének mérésére vezetem be. Ha egy osztály-alosztály kapcsolatban van polimorf metódus (a kiemelt döntések felületeit a polimorf metódusok adják), akkor az adott öröklődés döntésredundanciák megszüntetésére jött létre. A bevezetett metrikának a nagy értéke jó, vagyis minél több öröklődésben van polimorf metódus, a program szerkezete annál jobb.
A „Ekvivalens döntésesetek aránya", „Döntéspredikátumában eltérő egyéb viselkedésében egyező döntésesetek aránya", „Ekvivalens döntéspredikátumú döntésesetek aránya" metrikákkal a döntésredundanciák eseteinek teljesülése

mérhető. Ezek felhasználásával szofisztikáltabb képet kaphatunk a döntésszerkezetek állapotáról, melyekhez viszont a döntések viselkedési szerződésének elemzése szükséges.
(3.7. Döntésszerkezet elemzése új metrikákkal)

3.1 Döntés és döntéskiemelés

A program döntésszerkezete egy a program struktúrájától részben független tulajdonság, aminek szerkezete erősen jellemzi, hogy a program mennyire optimalizált. A szemléletesség illetve a megértés kedvéért a döntésszerkezetet elképzelhetjük döntés gráfként, ahol az egyes döntések a gráf élei, a csomópontok pedig a program állapotokat jellemzik. A döntés gráfot optimalizálhatjuk különböző transzformációkkal úgy, hogy a program viselkedése nem változik.
Hogy ezt a szerkezetet, szerkezet optimalizációs lépéseket/transzformációkat, illetve optimalizációs eseteket (amikor az optimalizációs lépések/transzformációk elvégzése indokolt) vizsgálni tudjuk szükséges a következő fogalmak bevezetése:

Döntés definíció:
A programban arról döntünk, hogy egy adott ponton milyen funkcionalitásra és/vagy adatszerkezetre van szükségünk.
Egy döntés döntéslehetőségekből áll. Amikor egy döntést definiálunk, a döntés döntéslehetőségeinek funkcionalitását és/vagy adatszerkezetét adjuk meg. A döntéslehetőségeket azok viselkedésével és adattartalmával (állapotleírásával) határozhatjuk meg. A döntéslehetőségek által specifikált viselkedés és adattartalom döntésfüggésben van.

Az egyes döntések olyan állapotátmenetek, amelyekkel az állapottér szűkebb lesz, azaz a lehetséges lefutások száma (elérhető állapotok száma) csökken és végül csak egy marad, hiszen a programunk determinisztikus (ugyanarra a bemenetre ugyanazt a kimenetet adja). A döntéslehetőségek a viselkedés (állapotsor) lehetséges irányait mutatják meg. Minden egyes döntéssel a döntéslehetőségben kiértékelt változókkal és az elérhető metódusokkal egy új állapotba kerülünk, amivel az állapotteret specializáljuk.

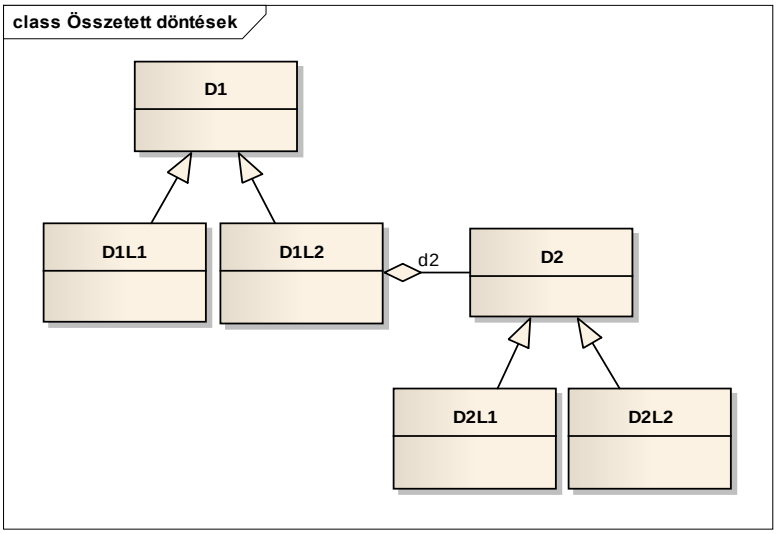
Az egyszerűség kedvéért minden döntés két döntéslehetőségből áll, azaz minden döntésfa kétélű. Mivel minden fa átalakítható bináris-kétélű fává, ezzel nem szűkítettük a vizsgálandó problémakört.
A döntés tartalmaz egy predikátumot (döntéspredikátum), amely alapján eldől, hogy a két döntéslehetőségből melyik kerül kiválasztásra. Egy döntés döntéslehetőségeire az adott döntés döntéspredikátuma vagy annak negáltja ($P_{D_1}, \neg P_{D_1}$) vonatkozik, vagyis az egyes döntéslehetőségeknek az adott döntés döntéspredikátuma illetve annak negáltja az előfeltétele.
Minden döntés esetén a döntéslehetőség „előfeltétele”, hogy az adott döntés döntéspredikátuma (P_{D_1}) az adott döntéslehetőségnek megfelelően igaz legyen.

Ha az adott döntéslehetőség teljesül, azaz pl. a P_{D_1} igaz, akkor az adott döntést „feloldottuk”, és az adott döntéslehetőség adattartalmával és a benne megadott metodológiával új állapot jön létre.
Egy döntéslehetőség akkor teljesül, ha a hozzá tarozó döntéspredikátum teljesül.
A döntések döntésesetei mindig metódusba „ágyazva” jelennek meg, ahol vagy az adott metódusban történik a döntés lehetőségeinek deklarálása egy feltételes utasítás 2 ága szerinti blokkba szervezve a döntéslehetőség adatstruktúráját, metodológiáját, vagy kiemelten egy objektumhivatkozással jelenik meg úgy, hogy a döntés definíciója egy osztály struktúrában kerül megadásra, és az adott döntésesetnél a döntéslehetőségek deklarációi nem látszódnak illetve az sem, hogy melyik döntéslehetőség érvényesül az adott esetben.

A döntések, döntéslehetőségek viselkedését és adatszerkezetét a következőképpen jellemezhetjük:
Döntéslehetőség viselkedése:
A teljesülő döntéslehetőség az általa specifikált viselkedéssel deklarálja az állapot átmenetet, vagyis azt, hogy a döntéslehetőség szerint a kiindulási állapoton milyen változásoknak kell történnie, ami egyrészt a korábbi állapotleírás adatszerkezetén végzett módosításokat jelenti, illetve a döntéslehetőség által deklarált új adatszerkezetek állapotleírásba való bevezetését, azaz az állapot további specializációját. A döntéslehetőség kiindulási állapotának kiemelt része a döntéspredikátum, ami az állapotátmenet előfeltétele.
A döntéslehetőség viselkedését, ha a döntés még nem kiemelt, akkor a feltételes utasítás két ága szerinti utasítások jelentik. Ha a döntés már kiemelt, akkor az osztályhierarchiába definiált alosztályok szerinti polimorf metódusok definiálják a döntéslehetőség szerinti állapotátmenetet jelentő viselkedést.

Döntéslehetőség adatszerkezete:
A teljesülő döntéslehetőség adatszerkezete adja meg, hogy döntéslehetőség viselkedése milyen adatszerkezeten értelmezhető, vagyis a döntéslehetőség kiinduló és végállapotának leírását biztosító adatszerkezet. A program állapotleírása kiegészül a teljesülő döntéslehetőség által létrejövő új adatleíró szerkezetekkel, illetve a döntéslehetőség adatszerkezetébe beletartozik a kiindulási állapot adatstruktúrájának azon része, amely adattartalma felhasználásra kerül a döntés során vagy az értéke változik. Tehát a döntéslehetőség adattartalmát, adatstruktúráját azon változók jelentik, melyeknek az adattartalma felhasználásra kerül és/vagy állapotuk módosul.
Ha a döntés még nem kiemelt, akkor a döntéslehetőség specifikus adatstruktúra a feltételes utasítás adott döntéslehetőséghez tartozó ágában felhasznált vagy módosított változók csoportját jelenti. Ha már kiemelt döntésről van szó, akkor az adott kiemelt döntésben, azaz osztályhierarchiában deklarálnunk kell az adott döntéslehetőségek által használt közös, illetve döntéslehetőség-specifikus változókat, melyek szintén szerepelhetnek akár csak adatforrásként, illetve az állapotváltozásokban is részt vehetnek.

Az egymásba épülő döntések esetén a döntéslehetőségek előfeltételét meghatározó döntéslehetőség döntéspredikátumok P_{D_1} logikai „és” kapcsolattal való összekapcsolásával kapjuk meg az egymásba épülő döntések előfeltételét. Pl.: $P_{D_1L_2} \wedge P_{D_2L_1}$...., amely a következő UML diagrammal [40] szemléltetett összetétel esetén mutatja egy lehetséges „viselkedésirány” előfeltételét:



1. Diagram: Objektum összetétel – Aggregáció – Döntések egymásba épülése

Két döntéspredikátum P_{D_1}, P_{D_2} akkor és csak akkor ekvivalens $P_{D_1} = P_{D_2}$, ha minden állapotban – ha kiértékeltek – az értékük megegyezik, azaz a döntéseket tartalmazó program viselkedése, a lehetséges állapotsorok, lefutások szerint a P_{D_1}, P_{D_2} döntéspredikátumok értéke mindig ekvivalens.

Két döntéspredikátum ekvivalens, ha azokat ugyanazon kifejezéssel reprezentáljuk, illetve, ha az egyik döntéspredikátum szerint megvalósuló döntés döntéslehetőségeiben kerül beállításra a másik döntés döntéspredikátuma vagy döntéspredikátum kifejezés részét képező operandusok.

Egy D döntés D_{L_1}, D_{L_2} döntéslehetőségei által elérhető viselkedések a döntéslehetőségek $C_{D_{L_1}}, C_{D_{L_2}}$ viselkedési szerződéseivel[21][28][44] adhatók meg (egy döntéslehetőség viselkedését egy viselkedési szerződéssel adhatjuk meg). D_{L_1}, D_{L_2} -n a döntéslehetőségek megvalósítását értjük, mellyel szinkronban deklarálhatók tehát a döntéslehetőségek követelményeit specifikáló $C_{D_{L_1}}, C_{D_{L_2}}$ viselkedési szerződések. A D_{L_1}, D_{L_2} által reprezentált döntésszerkezet megvalósítások változásai nem minden esetben jelentenek változásokat a $C_{D_{L_1}}, C_{D_{L_2}}$ viselkedési szerződéseken.

A döntéslehetőségek viselkedési szerződései deklarálják a döntéslehetőségek előfeltételeit, melyek a döntéslehetőségek döntéspredikátumai, a döntéslehetőségek utófeltételei pedig a döntéslehetőségek által eredményezett állapotátmeneteket határozzák meg. A viselkedési szerződés specifikálja az adatszerkezetet, amelyen a szerződés által meghatározott állapotátmenetek értelmezhetők. Egy típus viselkedési szerződésének állapotterét szűkítő invariánsokat [29] és állapot átmeneteit specifikáló történeti megszorításokat (history constraint [29]) a döntéslehetőségek elő- és utófeltételeinek részeként kezeljük, amelyek invariánsokként és történeti megszorításokként való értelmezése a döntéslehetőségek viselkedési szerződésének behatárolása szempontjából nem lényeges.

```
/**    Nem kiemelt döntés döntéslehetőségei.
*/
public void nemKiemeltDöntés(){
       if(döntéspredikátum){
              utasítás_1;
              …
              utasítás_i;

       }else {
              utasítás_j;
              …
              utasítás_k;

       }
}
```

A döntés döntéslehetőségeinek $C_{D_{L_1}}, C_{D_{L_2}}$ viselkedési szerződései a következő módon értelmezhetők:

$C_{D_{L_1}}$ - 1. döntéslehetőség viselkedési szerződése:

- Előfeltétel: *„döntéspredikátum”*
- Utófeltétel: *„utasítás_1, … , utasítás_i”* által végbement állapotátmenetet specifikáló logikai formula.
- Keretfeltétel: $C_{D_{L_1}}$ utófeltételei által érintett azon változók halmaza, melyek állapota változik.

$C_{D_{L_2}}$ - 2. döntéslehetőség viselkedési szerződése:

- Előfeltétel: *„!döntéspredikátum”*
- Utófeltétel: *„utasítás_j, … , utasítás_k”* által végbement állapotátmenetet specifikáló logikai formula.
- Keretfeltétel: $C_{D_{L_2}}$ utófeltételei által érintett azon változók halmaza, melyek állapota változik.

Döntéseset definíció:

A döntés egy előfordulása, amely során a döntés döntéspredikátumának kiértékelésével kiválasztásra kerül a megfelelő döntéslehetőség, amely által a

specifikált állapotátmenet végrehajtódik, illetve az állapotnak a döntéslehetőség által érintett adatszerkezetén az állapot átmenetnek megfelelően (állapotleírás vonatkozó része) a szükséges módosítások (állapotleírás egyes adatainak módosítása, állapotleírás kiterjesztése) megtörténnek.
Ha a döntéseseteket már kiemeltük és összevontuk, azaz a döntést kiemeltük (lásd: Döntéskiemelés), akkor megkülönböztetünk inicializáló döntésesetet, majd az inicializáláskor archivált döntés későbbi felhasználásának helyeit.
Kiemelt döntés esetén az inicializáló döntésesetnél a döntéspredikátum kiértékelésével meghatározott döntéslehetőség, mint a döntés eredménye archiválásra kerül a kiemelt döntéshez tartozó osztálystruktúra szülő osztály típusú referenciával, amely az inicializáláskor meghatározott döntéslehetőség szerinti alosztály egy példányára mutat.
Az adott döntés további döntéseseteinél az első döntéseset eredményét használjuk fel (a döntéslehetőség elérését biztosító polimorf metódusmeghívással), vagyis nem kell újra kiértékelni a döntést.
Összefoglalva döntésesetnek nevezzük azt a helyet, ahol a döntés tényleg megszületik és kiemelt döntések esetén archiválódik, illetve a későbbi felhasználások helyét is, amelyeknél nem kiemelt döntések esetén újra végre kell hajtani a döntéseket, kiemelt döntések esetén azonban az archivált döntések újra felhasználhatók, azaz a döntések újra kiértékelésére nincs szükség.

Döntésfüggőség definíció:
Döntésfüggőségen a döntésszerkezeti implementációs függőséget értjük. Akkor beszélhetünk döntésfüggőségről, ha egy döntés döntéslehetőségeinek – azok viselkedési szerződésének – változása, vagy egy új döntéslehetőség bevezetése a döntéseseteknél változtatáskényszert eredményez. Osztályhierarchiába kiemelt döntések használatakor, ha a döntéslehetőségek viselkedése változik, vagy új döntéslehetőséget vezetünk be, akkor nem szükséges az egyes döntéseseteken változtatni, csak az inicializáló döntésesetnél lehet változás.
Az öröklődés alkalmazása akkor jogos, ha a döntésszerkezeti függőségek indokolják.

Döntéskiemelés definíció:
A döntésfüggőségben lévő programrészek függőségeit döntéskiemeléssel tudjuk megszűntetni úgy, hogy a döntés döntéslehetőségeink viselkedését és adatszerkezetét nem egy feltételes utasítás ágaiban, hanem egy osztályszerkezetben adjuk meg, ahol az alosztályokban deklaráljuk a döntés döntéslehetőségeit, amelyeket a szülőosztály fog össze. A döntés „felületét” egy a szülőosztályban definiált polimorf metódus jelenti, amit az alosztályok felülírnak.
A kiemelés után a döntéseseteknél, azaz a döntési helyeken a feltételes utasítás helyett a döntés az azt deklaráló osztályszerkezet szülőosztálya szerinti típusú referenciával jelenik meg, ami az eset szerinti döntéslehetőséghez tartozó alosztály egy példányára mutat.
Legyen D_{NK} egy nem kiemelt döntés, amely $D_{NK\,L_1}, D_{NK\,L_2}$ döntéslehetőségei teljesítik a $C_{D_{NK\,L_1}}, C_{D_{NK\,L_2}}$ viselkedési szerződéseket. A D_K a D_{NK} kiemelt döntése,

ha a D_K döntés D_{KL_1}, D_{KL_2} döntéslehetőségei által teljesülő a $C_{D_{KL_1}}, C_{D_{KL_2}}$ viselkedési szerződésekre igaz a következő:
$C_{D_{KL_1}} = C_{D_{NKL_1}}$, $C_{D_{KL_2}} = C_{D_{NKL_2}}$, azaz a nem kiemelt és kiemelt döntéslehetőségek viselkedési szerződései ekvivalensek.

3.2 A döntésredundanciák elkerülésének esetei

Az előző fejezetben taglaltam a döntéskiemelés módszerét, mellyel egy döntés újrafelhasználható módon deklarálható. Ugyanakkor fontos észrevenni, hogy a kiemelés abban az esetben indokolt, ha azzal várható vagy már meglévő döntésismétlődés/döntésredundancia kerülhető el, amely implementációs függőségeket eredményez csökkentve a kód átláthatóságát és karbantarthatóságát.
A következőkben azon szabályokat fogalmazom meg, amelyek betartásával a döntésredundancia elkerülhető.

A döntésredundancia elkerülésének feltételei a következők:

- A döntésesetek ne ismétlődjenek, azaz ugyanazt a döntést egy lefutás alatt ne kelljen többször meghoznunk. Ehhez szükséges, hogy a döntéseknek megfelelő viselkedés/metodológia, állapotleírás/adatszerkezet csak egy helyen legyen definiálva, azaz ne legyen a döntéslehetőségeknek megfelelő viselkedés (döntéspredikátumok is ekvivalensek a viselkedés részeként) vagy/és állapotleírás, adatszerkezet többször megadva. Tehát ugyanaz a döntés és annak körülményei a programban ne szerepeljenek többször, kivéve, ha az adott eset nem teszi lehetővé a döntésösszevonást.
- Nem egyező döntéspredikátumú, de egyéb tulajdonságaiban egyező döntés és annak körülményei a programban ne szerepeljenek többször, kivéve, ha az adott eset nem teszi lehetővé a döntések összevonását. A döntéseknek megfelelő viselkedés/metodológia, állapotleírás/adatszerkezet a döntéspredikátumot, mint a viselkedés részét képező előfeltételt kivéve csak egy helyen legyen definiálva, azaz ne legyen a döntéslehetőségeknek megfelelő viselkedés vagy/és állapotleírás, adatszerkezet többször megadva.
- Ekvivalens döntéspredikátumú döntések, melyek nem egyező adattartalmat-adatszerkezetet, metodológiát-viselkedést definiálnak, ne kelljen többször meghoznunk. Ehhez szükséges, hogy a döntéseket összevontan, egy helyen definiáljuk.

3.3 Döntésösszevonás

A döntésredundanciák feloldásának eszköze a döntésösszevonás. A döntésösszevonás lehetőségeit a döntésredundanciák esetei alapján határozom meg.
Felhasználom a Liskov féle helyettesítési elvre alapozott altípus-szülőtípus közötti helyettesíthetőséggel kapcsolatosan megfogalmazott a viselkedési szerződések

viszonyára vonatkozó meghatározást [44], miszerint C' viselkedési szerződés a C viselkedési szerződés erősítése, értelmezésem szerint **valódi finomítása**, azaz $C' \supset C$ (felolvasva: C' viselkedési szerződésből a C viselkedési szerződés következik), ha a C' további megkötéseket specifikál, de teljesíti a C viselkedési szerződés megkötéseit. A viselkedési szerződések utófeltételei tekintetében ez „erősítést", azaz további feltételek megadását, előfeltételek tekintetében „gyengítést" jelent. Az előfeltételként értelmezett döntéspredikátum nem változhat.
A C' viselkedési szerződés a C viselkedési szerződés **finomítása**, azaz $C' \supseteq C$ (felolvasva: C' viselkedési szerződésből a C viselkedési szerződés következik vagy a kettő egyezik), ha viselkedési specifikációjukban megegyeznek vagy a C' további megkötéseket specifikál, de teljesíti a C viselkedési szerződés megkötéseit. A viselkedési szerződések utófeltételei tekintetében ez egyezőséget/„erősítést", azaz a feltételek megtartását/további feltételek megadását, előfeltételek tekintetében egyezőséget/„gyengítést" jelent. Az előfeltételként értelmezett döntéspredikátum nem változhat.

Döntésösszevonás definíciója:
Észre kell vennünk, hogy a nem kiemelt és kiemelt döntések viselkedési szerződései között lehetnek részleges vagy teljes egyezőségek, amelyek esetén a döntésredundanciák elkerülése érdekében szükséges a döntések részleges vagy teljes összevonása.
Fontos megjegyezni, hogy a nem kiemelt döntések esetén a döntésösszevonás úgy lehetséges, hogy az adott döntéseket döntéskiemeléssel osztály-alosztály szerkezetbe kiemeljük, amely után már végrehajtható az összevonás.
Összegezve tehát a döntéskiemeléssel nyílik lehetőség a nem kiemelt döntések esetén a döntésösszevonások végrehajtására.
Ahhoz, hogy megállapíthassuk, hogy a két döntés összevonható, a két döntés viselkedési szerződését kell összehasonlítanunk, amely alapján eldől, hogy a döntésösszevonás valamely esetének megfelelnek-e a vizsgált döntések.
Fontos megjegyezni, hogy a döntésösszevonásra már kiemelt döntések esetén is szükség lehet, tehát a döntésösszevonásoknak vannak olyan esetei, ahol már kiemelt döntéseket kell összevonni.
A döntésredundancia elkerülésének előzőekben ismertetett szabályai szerinti esetekre bontva a következőkben megadott feltételek teljesülése esetén indokolt a „döntésösszevonás"/"rész döntésösszevonás":

1. **Teljesen vagy részben egyező döntések összevonása/részösszevonása:**
   Döntésösszevonás/döntés részösszevonás indokolt, ha a döntések döntéspredikátumai ekvivalensek, illetve döntéslehetőségei olyan adattartalmat és funkcionalistást deklarálnak, amelyek egyeznek, részben egyeznek, egyik a másik kiterjesztése, egy közös résznek mindkettő kiterjesztése. Nyilván, ha a vizsgált döntések már kiemeltek, de az összevonás kritériumai teljesülnek, akkor is el kell végezni az összevonást.

Ekvivalens vagy egyik a másik döntést viselkedésileg kiterjesztő döntések összevonása:

Két döntés/döntéseset a következő esetekben vonható össze:

- Ha a két döntés/döntéseset döntéslehetőségei által nyújtott viselkedést megadó szerződések megegyeznek.
- Ha az egyik a másik által nyújtott viselkedést (viselkedési szerződést) finomítja, erősíti.

Legyenek D_1, D_2 olyan döntések, amelyek D_{1L_1}, D_{1L_2} és D_{2L_1}, D_{2L_2} döntéslehetőségei teljesítik a $C_{D_{1L_1}}, C_{D_{1L_2}}, C_{D_{2L_1}}, C_{D_{2L_2}}$ viselkedési szerződéseket. A D_1, D_2 döntések összevonhatók, ha léteznek olyan $C_{D_{L_1}}, C_{D_{L_2}}$ viselkedési szerződések, melyekre a következő teljesül:

$$C_{D_{L_1}} = C_{D_{1L_1}}, C_{D_{L_1}} \supseteq C_{D_{2L_1}}, C_{D_{L_2}} = C_{D_{1L_2}}, C_{D_{L_2}} \supseteq C_{D_{2L_2}}$$

Tehát, ha az összevonandó döntések egyikének viselkedési szerződése ekvivalens vagy finomítja-erősíti a másik döntés viselkedési szerződését, azaz az összevont döntés viselkedési szerződése az egyik összevont döntés viselkedési szerződésével ekvivalens, a másik döntés viselkedési szerződésének finomítása.

Ennek megfelelően vizsgálni kell, hogy a döntések adatstruktúrája, illetve viselkedése, azaz a döntésfüggő adatstruktúrák és viselkedések egyeznek-e, figyelve arra, hogy esetlegesen nem teljes egyezés van, hanem az egyik finomítása a másiknak.

Eszerint a döntések akkor vonhatók össze, ha az adatszerkezetük, illetve viselkedésük tekintetében a következők teljesülnek:

- Adatszerkezet: A döntések állapotleíró szerkezete megegyezik, vagy egyik finomítása a másiknak. Ezek szerint a következő eseteket határozhatjuk meg:
  - A döntések változói által elérhető állapotok halmaza egyezik.
  - Az egyik döntés adatszerkezetét adó változókkal meghatározható állapotok részhalmazát képezik a másik döntés változóival meghatározható állapotok halmazának. („Díszítő” tervezési minta használata egyik döntésre)
- Viselkedés: Ha a két döntés döntéslehetőségei által megadott viselkedések elő és utófeltételei egyeznek, illetve az egyik döntés a másik döntés viselkedését deklaráló utófeltételhez képest ad meg további megszorításokat, és/vagy előfeltételét szűkíti. Eszerint a viselkedések szempontjából elérhető lefutások halmaza egyezik, az egyik a másiknak részhalmaza (egyik kiterjeszti a másikat).

Döntés részösszevonása:

Két döntés részösszevonása akkor lehetséges, ha a döntések viselkedési szerződéseinek van elhatárolható közös része, melyet mindkét döntés további feltételekkel egészít ki, melyeket nem vonunk össze.

Legyenek D_1, D_2 olyan döntések, amelyek D_{1L_1}, D_{1L_2} és D_{2L_1}, D_{2L_2} döntéslehetőségei teljesítik a $C_{D_{1L_1}}, C_{D_{1L_2}}, C_{D_{2L_1}}, C_{D_{2L_2}}$ viselkedési szerződéseket. A D_1, D_2 döntések részlegesen összevonhatók, ha léteznek olyan $C_{D_{L_1}}, C_{D_{L_2}}$ viselkedési szerződések, melyekre jellemzők az alábbiak:

$$C_{D_{L_1}} \subset C_{D_{1L_1}}, C_{D_{L_1}} \subseteq C_{D_{2L_1}}, C_{D_{L_2}} \subseteq C_{D_{1L_2}}, C_{D_{L_2}} \subseteq C_{D_{2L_2}},$$

azaz a D_1 döntés döntéslehetőségei által megfogalmazott viselkedési szerződéseknek és a D_2 döntés döntéslehetőségek viselkedési szerződéseinek vesszük a „közös részét” ($C_{D_{L_1}}, C_{D_{L_2}}$), amit összevonunk. Az így létrejövő D szerződés döntéslehetőségek $C_{D_{L_1}}, C_{D_{L_2}}$ viselkedési szerződéseinek a D_1, D_2 döntéslehetőségek viselkedési szerződései valódi finomításai, finomításai úgy, hogy legalább egy döntéslehetőség viselkedési szerződése esetén a valódi finomítás áll fenn. (Ha minden viselkedési szerződés esetén megengednénk a finomítást, akkor olyan eseteket is a döntés részösszevonásba sorolnánk, mikor minden összevonandó döntéslehetőség viselkedési szerződése ekvivalens. Ezt az esetet azonban a döntésösszevonások közé soroljuk.)
Kijelenthető, hogy az elhatárolható közös rész a két viselkedési szerződés metszete, azaz a következőknek teljesülni kell:

$$C_{D_{1L_1}} = C_{D_{L_1}} \wedge C_{D_{1L_1}egyedi}, C_{D_{1L_2}} = C_{D_{L_2}} \wedge C_{D_{1L_2}egyedi}$$

$$C_{D_{2L_1}} = C_{D_{L_1}} \wedge C_{D_{2L_1}egyedi}, C_{D_{2L_2}} = C_{D_{L_2}} \wedge C_{D_{2L_2}egyedi},$$

ahol a $C_{D_{1L_1}egyedi}, C_{D_{1L_2}egyedi}, C_{D_{2L_1}egyedi}, C_{D_{2L_2}egyedi}$ viselkedési szerződések határozzák meg az összevonandó döntések kiemelt részt kiegészítő döntés-specifikus viselkedéseit.

Ennek megfelelően vizsgálni kell, hogy a döntések adatstruktúrája, illetve viselkedése egy meghatározott közös résznek finomítása, valódi finomítása. Eszerint a döntések akkor vonhatók össze, ha az adatszerkezetük illetve viselkedésük tekintetében a következők teljesülnek:

- Adatszerkezet: A döntések állapotleíró szerkezete megegyezik, vagy egyik valódi finomítása a másiknak, vagy mindkettő valódi finomítása, kiterjesztése egy közös állapot halmaznak. Tehát a döntésváltozókkal elérhető állapothalmazoknak metszetük van, azaz mindkettő döntés által elérhető állapothalmaz kiterjesztése egy közös résznek. (Díszítő tervezési minta alkalmazása mindkét döntésre)
- Viselkedés: Ha a két döntés döntéslehetőségei által megadott viselkedések elő és utófeltételei egy közös viselkedést deklaráló utófeltételekhez képest adnak meg további megszorításokat, és/vagy a közös viselkedést deklaráló előfeltételeket szűkítik.

Eszerint a viselkedések szempontjából elérhető lefutások halmazainak van metszete (a közös részt mindkettő kiterjeszti).

2. **Nem egyező döntéspredikátumú, de egyéb tulajdonságaiban egyező döntések összevonása/részösszevonása:**
   Döntésösszevonás/döntés részösszevonás indokolt, ha a döntések döntés-predikátumai nem ekvivalensek, de döntéslehetőségeik olyan adattartalmat és funkcionalistást deklarálnak, amelyek egyeznek, részben egyeznek, egyik a másik kiterjesztése, egy közös résznek mindkettő kiterjesztése. Nyilván, ha a vizsgált döntések már kiemeltek, de az összevonás kritériumai teljesülnek, akkor is el kell végezni az összevonást.
   Vizsgálni kell, hogy a döntések adatstruktúrája és viselkedése a döntéspredikátumokat, mint előfeltételeket kivéve egyeznek-e, figyelve arra, hogy esetlegesen nem teljes egyezés van, hanem az egyik finomítása a másiknak, illetve egy meghatározott közös részt mindkettő finomítja.
   Az összevonás és részösszevonás szabályai az előbb taglalt feltételek kivételével az előző esettel egyeznek.

3. **Ekvivalens döntéspredikátumú, de döntéslehetőségeik viselkedési szerződéseiben eltérő döntések összevonása:**
   Ha két akár kiemelt döntés döntéspredikátumai ekvivalensek (adatszerkezetük és viselkedésük nem vizsgált), akkor összevonhatóak, mellyel a két döntés szerinti adatszerkezet és viselkedés „összeolvad”.
   Tehát a döntések összevonhatók, ha adatszerkezetük nem egyezik és viselkedésüket adó elő- és utófeltételek közül csak az előfeltételeket jelentő döntéspredikátumok egyeznek.
   A döntések döntéspredikátumai ekvivalensek: $P_{D_1} = P_{D_2}$, ha a program viselkedése alapján, tehát az azt megadó lehetséges állapotsorok, azaz minden lefutás esetén a kiértékelt döntéspredikátumok ekvivalensek. A döntéspredikátum akkor kiértékelt, ha a döntéspredikátumot deklaráló kifejezés kiértékelt.
   Legyenek D_1, D_2 olyan döntések, amelyek D_{1L_1}, D_{1L_2} és D_{2L_1}, D_{2L_2} döntéslehetőségei teljesítik a $C_{D_{1L_1}}, C_{D_{1L_2}}, C_{D_{2L_1}}, C_{D_{2L_2}}$ viselkedési szerződéseket. A D_1, D_2 döntések összevonhatók, ha léteznek olyan $C_{D_{L_1}}, C_{D_{L_2}}$ viselkedési szerződések, melyek az összevont döntések viselkedési szerződéseinek diszjunktciói:

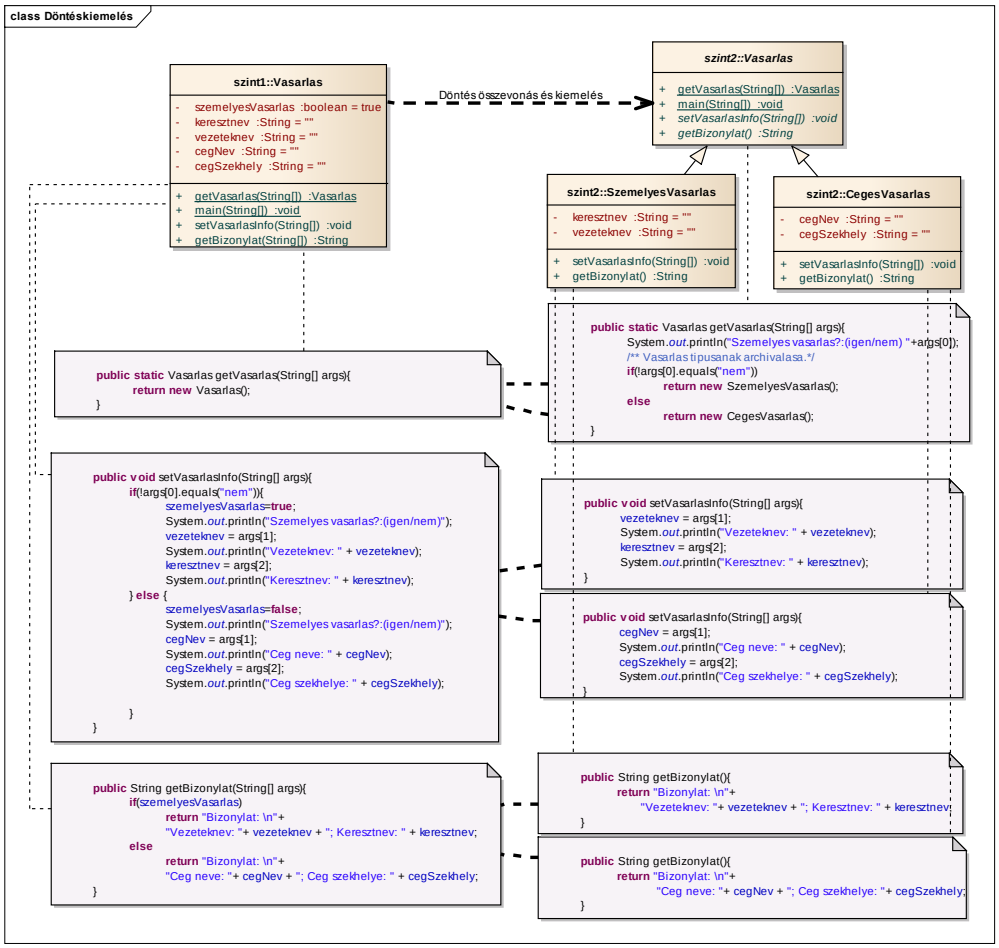
$$C_{D_{L_1}} = C_{D_{1L_1}} \vee C_{D_{2L_1}},$$

$$C_{D_{L_2}} = C_{D_{1L_2}} \vee C_{D_{2L_2}}$$

   (A „5.3.2 Példa – Híd tervezési minta döntéskiemelés előtt és után” fejezetben bemutatott példaprogramban szereplő döntések összevonása és kiemelése ezt az esetet mutatja be!)

Az objektumorientált programozási módszertan egyik lényeges célja nem más, mint a programban lévő döntésismétlődések, döntésredundanciák megszüntetése, a döntések összevonásával és kiemelésével azok egy strukturáltabb formában történő deklarálásával.

3.4 Döntéskiemelés, döntésösszevonás példa



2. Diagram: Döntéskiemelés osztályhierarchiába

A példából látható, ahogy egy döntéskiemeléssel két döntés döntéslehetőségeit osztályhierarchiába szervezzük, aminek segítségével lehetőség van arra, hogy a döntéseket egy szülőosztály típusú objektumba zárva (*Vasarlas*) többször felhasználjuk, először a döntésnek megfelelő adatok lekérdezésére (*vasarlas.setVasarlasInfo()*), majd azok döntésnek megfelelő megjelenítésére (*vasarlas.getBizonylat()*).

Egy osztályhierarchiába kiemelt döntés esetén az adott osztály vagy valamelyik alosztály példányosításakor eldől, hogy a döntés mely iránya kap „zöld utat”, vagyis

hogy melyik döntéslehetőség szerinti adattartalom tagváltozóinak inicializálása történik meg, melyik metodológiája válik elérhetővé.
Az 2. diagramon a kiemelés utáni állapotban a következő kódrészletben történik a példányosítás:

```
System.out.println("Szemelyes vasarlas?:(igen/nem) "+args[0]);
/** Vasarlas tipusanak archivalasa.*/
if(!args[0].equals("nem"))
        return new SzemelyesVasarlas();
else
        return new CegesVasarlas();
```

Személyes vásárlás esetén a *SzemelyesVasarlas* alosztályban megadott *vezeteknev*, *keresztnev* változók inicializálódnak, illetve a *SzemelyesVasarlas* alosztály specifikus *setVasarlasInfo()*, *getBizonylat()* metódusok válnak elérhetővé.
Céges vásárlás esetén a *CegesVasarlas* alosztályban megadott *cegNev*, *cegSzekhely* változók inicializálódnak, illetve a *CegesVasarlas* alosztály specifikus *setVasarlasInfo()*, *getBizonylat()* metódusok válnak elérhetővé.
Az adott döntés döntéspredikátuma:

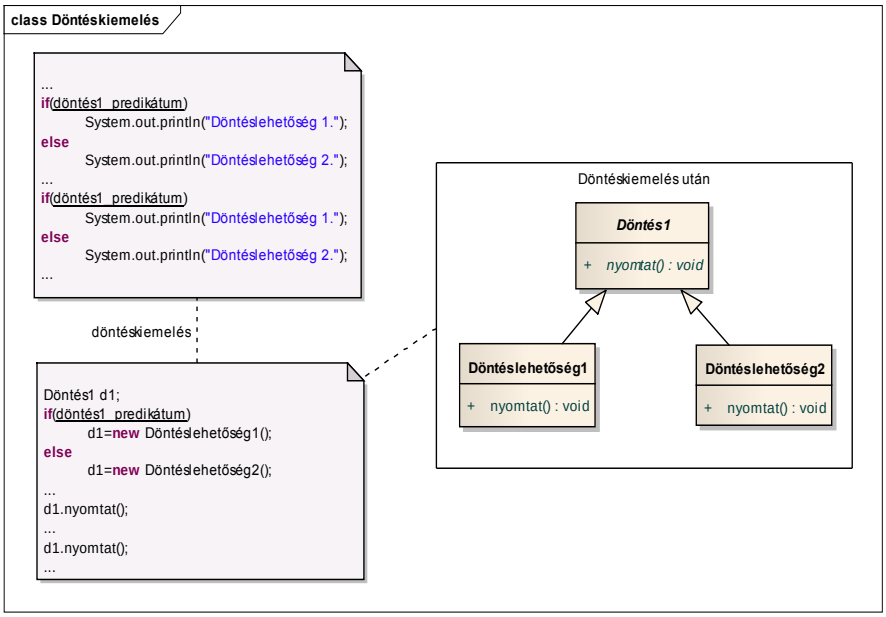
```
!args[0].equals("nem")
```

A döntéspredikátum értékelése alapján határozódik meg, hogy melyik döntéslehetőségnek megfelelő alosztály szerinti metodológiával és adattartalommal fog az adott referenciaváltozó inicializálódni, azaz melyik alosztály szerinti objektumra fog a referencia mutatni.

3.5 Objektumorientált paradigmák kiegészítése

Hogy a tervezés, refaktorálás során a döntésszerkezet szerinti elemzést el tudjuk végezni, tudnunk kell, hogy az objektumorientált programozás alapfogalmai (származtatás-öröklődés [5][11][39][41], polimorfizmus [5][11][39], egységbezárás [5][11][39]), illetve alapeszközei (osztályhierarchia [15], objektum összetétel [15], aggregáció/képviselet [15]) a döntésszerkezetekhez hogyan viszonyulnak.
Öröklődés, mint döntés absztrakció:
Egy döntés esetén azt határozzuk meg, hogy egy adott ponton milyen programrész fusson le, milyen struktúrájú adattartalom szerepeljen. A döntés kiemelése során a döntéslehetőségek körülményeit (metodológia és/vagy adattartalom) alosztályokba emeljük ki. Az öröklődés nem más, mint egy döntésabsztrakció, vagyis, ha döntésredundancia lép fel („3.2 A döntésredundanciák elkerülésének esetei”), akkor annak kiküszübölése a döntésösszevonás szabályainak alkalmazásával („3.3 Döntésösszevonás”), az összevonás részeként az adott döntés(ek) és körülményei(k)nek egy osztályhierarchiába emelésével tehető meg, ahol a döntést a szülőosztályban definiált polimorf metódus fogja képviselni.
Egy osztályhierarchiába kiemelt döntés megvalósítja a következőket:

- Döntéslehetőségeket definiáló kód redundanciájának megszüntetése. (Döntéslehetőségek körülményeit csak egyszer definiáljuk az adott származtatással.)



3. Diagram: Kód redundancia megszüntetés kiemeléssel

Jól látható, hogy a *Döntés1* döntést döntéslehetőségeivel kétszer adtuk meg, amely döntésredundanciát jelent. Ezt a döntések *Döntés1* osztályhierarchiába történő kiemelésével szüntettük meg, amely után a döntést egy a *Döntés1* típusú *d1* referenciával archiváljuk. A *d1* referencia használható fel a későbbi döntéseseteknél, ahol a döntések már bezárva jelennek meg.

- Új alosztály bevezetésével a döntési lehetőségek könnyen bővíthetők lesznek úgy, hogy a későbbi felhasználások során nem kell módosítani a működést (a döntés „bezárás” miatt). Kivéve, ha döntéslehetőség specifikus adattartalomra vagy viselkedésre van szükség.

Polimorfizmus, mint döntésbezárás:

Ha egy döntést meghozva egy adott döntésesetben a megfelelő döntéslehetőség szerinti alosztályt példányosítjuk, az így kapott objektum úgy archiválja a döntést, hogy azt a későbbiekben egy szülőosztály típusú polimorf referenciával hivatkozzuk. Ezzel megvalósul a döntés „bezárása”, mivel a későbbi felhasználás során nem csak hogy nem kell a döntést újra meghozni, de a polimorf működés következtében azt sem kell tudnunk, hogy a döntésnek mi az eredménye, ami pedig meghatározza, hogy a későbbi esetekben milyen döntéslehetőségnek megfelelő metodológia és adatstruktúra fog érvényesülni. Tehát a döntéskiemelés után a döntést reprezentáló polimorf metódus segítségével valósul meg a döntésbezárás a döntésesetek során.

Az 3. diagramon bemutatott kiemelt döntésnél jól látható, hogy a döntést a *d1* referencia miként archiválja azzal, hogy a döntésnek megfelelő alosztály (*Döntéslehetőség1, Döntéslehetőség2*) egyedére mutat.

A példányosítás és a döntésbezárás a következő kódrészben történik:

```
if(döntés1_predikátum))
        d1= new Döntéslehetőség1();
else
        d1= new Döntéslehetőség2();
```

A későbbi felhasználások, metódushívások (*d1.nyomtat()*), hogy ténylegesen melyik metódus futását fogják eredményezni (melyik alosztály szerinti metódus fog lefutni) az határozza meg, hogy melyik alosztály példányára mutat a *d1* referencia. A *d1* referenciával archivált döntést a későbbiekben kétszer használjuk fel a *d1.nyomtat()* polimorf metódushívásokkal a későbbi döntéseseteknél, ahol a döntések már bezárva jelennek meg, hiszen ezt a polimorf működés biztosítja.
Jól érzékelhető tehát, hogy a polimorfizmus miként biztosítja a döntések bezárását, hiszen a példányosítást követő metódushívásoknál – amelyek a kiemelés előtt döntésesetek voltak – nem is lehet tudni, hogy melyik döntéslehetőségnek-alosztálynak megfelelő metódus fog lefutni, mivel a kezdeti döntésesetnél lévő döntéslehetőség szerinti példányosítással és szülő osztály típusú referenciával a döntés bezárás megvalósul, és a későbbi helyeken a többalakúság a döntéslehetőségeknek megfelelő különböző működést jelenti.

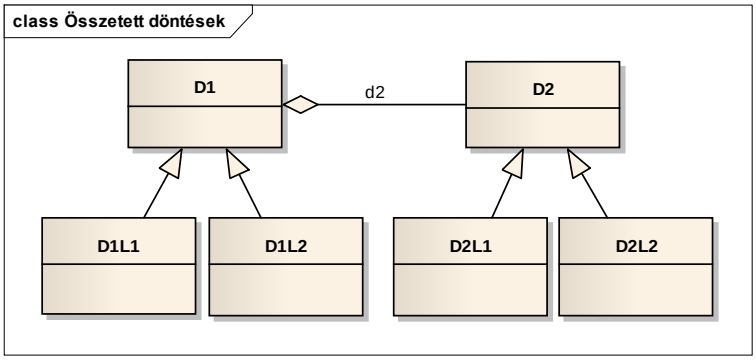
Egységbezárás, mint döntéslehetőség adatstruktúra és viselkedés egységben kezelése:
A döntés lehetőségeit adatstruktúrákkal és a ráépülő metodológiával tudjuk megadni. Ha az adott döntés még nem kiemelt formában, azaz egy metódusban definiált, akkor az adott feltételes utasítás megfelelő blokkjában van megadva az adatstruktúra és viselkedés.
A „3.4 Döntéskiemelés, döntésösszevonás példa” fejezetben a 2. diagramon a *CegesVasarlas* osztályban van megadva a céges vásárlás döntéslehetőség által érintett adatstruktúra: *cégNév, cégSzékhely* és metodológiája: *setVasarlasInfo(), getBizonylat()*.

Bezárás/információ elrejtés, mint döntésszerkezet bezárása:
Döntésszerkezetek kezelése során a döntés bezárás a döntések osztályhierarchiába történő kiemelésével oldható meg, ami után a döntést az osztályhierarchia szülőosztályában/interfészében deklarált polimorf, általában absztrakt metódussal képviselhetjük a felhasználási helyeken úgy, hogy nem szükséges tudnunk, hogy az egyes döntéslehetőségek milyen megvalósítással rendelkeznek, melyek így cserélhetők.

Objektum összetétel, aggregáció/képviselet, mint dinamikus döntésbeágyazás:
Egy referenciával megvalósuló objektum összetétellel vagy aggregációval egy kiemelt döntésre hivatkozhatunk. Abban az esetben, ha egy döntéseset során dönteni kell, hogy melyik döntéslehetőséget válasszuk (milyen adattartalommal, milyen metodológiával), illetve a későbbiekben az adott döntés tükrében (döntésismétlődés) műveleteket kell végezni a döntésnek megfelelő adattartalmon, akkor referenciával megvalósuló objektum összetétellel vagy aggregációval tudjuk az adott ponton a kiemelt döntést „képviselni”. („3.4 Döntéskiemelés, döntésösszevonás példa” fejezet 2. diagram *vasarlas* objektum, 3. diagram *d1*

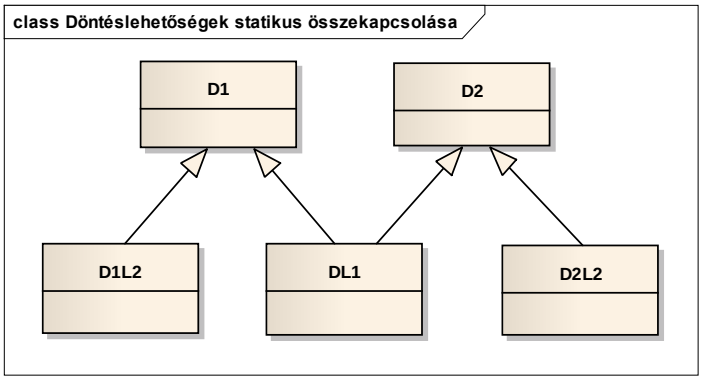
objektum) Ezzel a döntést archiváljuk és így a döntések későbbi felhasználhatóságát biztosítjuk, megszüntetve a döntésismétlődést.
A *Döntés1* esetén jól látható, hogy *Döntés1* szülőosztály típusú *d1* referenciával döntés miként archiválódik. A *d1* referenciával hivatkozott objektum valamelyik alosztály egy egyede, amely archiválja a döntést, megadva a döntésesetnek megfelelő döntéslehetőség adatstruktúrát és metodológiát.

Ha összetett döntésről van szó, ahol egy döntést definiálunk – *D1*, amelyben van egy tagváltozó, amely típusa egy másik osztályhierarchia-döntés – *D2* szülőosztálya, akkor az alapvetően nem más, mint egy döntés – *D1*, döntéslehetőség előfeltételének kiegészítése „és” kapcsolattal egy másik döntés – *D2*, döntéslehetőségének előfeltételével.



4. Diagram: Kiemelt döntések összekapcsolása aggregációval/objektum összetétellel

Többszörös öröklődés, mint döntéslehetőségek összekapcsolása:
Lehetséges, hogy különböző döntéseknek vannak olyan döntéslehetőségei, melyek döntéspredikátumai ekvivalensek. Lehetséges, hogy a döntéslehetőségek viselkedése, azaz adatstruktúrája és funkcionalitása teljesen vagy részben egyezik. Ebben az esetben az adott döntéslehetőségek „összevonhatók”, azaz a két döntéslehetőség definíciója megadható együtt. Ezt hívjuk a döntéslehetőségek statikus összekapcsolásának, ami úgy tehető meg, hogy már osztályhierarchiába kiemelt döntések egy-egy döntéslehetőségét megadó osztályokat összevontan adjuk meg, amely osztály többszörös öröklési kapcsolatban van a két döntés szülőosztályával. Ebben az esetben a két döntés összekapcsolt döntéslehetőségeire vonatkozó döntéspredikátumok ekvivalensek.
Ennek alternatívája lehet, ha ilyen esetben az egyik döntés lehetőség aggregációval hivatkozza a másikat. (Lásd: „*Illesztő*” tervezési minta [15])



5. Diagram: Döntéslehetőségek összekapcsolása többszörös örökléssel

3.6 Objektumorientált tervezési alapelvek kiegészítése

Nem megfelelően implementált, redundáns döntések implementációs függőségeket eredményeznek, melyeket döntésszerkezeti implementációs függőségnek nevezek.
Az előző fejezetekben bemutattam, hogy az öröklődés alkalmazása, azaz a döntés alosztályokba történő kiemelése, illetve a kiemelt döntéseken végrehajtható döntésösszevonások a döntésredundanciák megszüntetésének eszközei, melyekkel elkerülhető, hogy egy döntés körülményeinek megváltoztatása több komponens módosítását eredményezze. Tehát a döntésszerkezeti implementációs függőség a megfelelő döntésszerkezet alkalmazásával, azaz döntéskiemeléssel és összevonással feloldható.
A következőkben az implementációs függőségek elkerülésének módszereiként bevezetett objektumorientált programozási és tervezési alapelvek („2.2 Az objektumorientált tervezés alapelvei”) értelmezését egészítem ki a döntésszerkezetek szemszögéből közelítve meg azokat.

Programozzunk felületre:
A „Programozzunk felületre” elv a döntés struktúra szempontjából azt jelenti, hogy a döntéseket, ahol szükséges, ki kell emelni egy osztályszerkezetbe, ahol a szülőosztályban lévő polimorf metódus fogja ábrázolni az adott döntés „felületét”. Tehát a „Programozzunk felületre” elv döntésszerkezetre értelmezett változata az, hogy az összevonható döntéseket összevonással és kiemeléssel rendezzük közös felület alá.
Ha a szülőosztályban a döntés felületét adó metódus absztrakt metódus, amit csak az alosztályok implementálnak, és adják meg a döntés specifikus viselkedést, akkor nincs az adott döntés lehetőségeinek közös viselkedése. Ha a döntés felületét adó polimorf metódus nem absztrakt, akkor a döntésnek lehetnek döntés független közös részei, amelyek mindegyik döntéslehetőségre vonatkoznak.
Abban az esetben, ha a döntéshierarchia szülőosztálya nem absztrakt, azaz példányosítható, akkor lehetséges, hogy a döntést reprezentáló osztálystruktúra adott esetben nem tárol döntést, azaz nem csak a polimorf referenciának típusa a szülőosztály, hanem a hivatkozott objektum is szülőosztály típusú.

Ha megköveteljük, hogy a referencia mindenképp tároljon döntést, akkor a döntést reprezentáló osztályszerkezet szülőosztályát absztrakt osztályként kell deklarálni, melyet így nem lehet példányosítani.

Objektum összetétel használata öröklés helyett, ahol lehet:
A cél, hogy a két eszköz (öröklés, objektum összetétel) felhasználásával egy harmonikus osztály és objektumszerkezet keletkezzen. Az öröklés egy nagyon erős kötést jelent két osztály között, melyet használni csak indokolt esetben érdemes.
A döntésösszevonás szabályai („3.3 Döntésösszevonás”) szerint egy osztály-alosztály szerkezet, azaz öröklés használata akkor indokolt, ha vannak összevonható és ennek megfelelően szülő-alosztály szerkezetbe kiemelhető döntések, amellyel absztraktabb, azaz optimálisabb döntés reprezentáció érhető el, illetve, mellyel döntésredundanciát szűntetünk meg.

Liskov féle helyettesíthetőség elve – LSP:
A döntéskiemelés és döntésösszevonás szabályai szerint létrejövő osztályszerkezetben reprezentált döntések esetén az osztály-alosztály kapcsolatok egyben típus-altípus kapcsolatok is, azaz a döntéslehetőségek szerinti alosztályok viselkedése csak kiterjeszti a szülőosztály szerinti viselkedési követelményeket. A szülőosztályok specifikálják a döntések döntéslehetőségeinek közös viselkedését, amelyet a döntéslehetőségek csupán a döntéslehetőség szerinti speciális viselkedéssel egészítenek ki. Ezek szerint kijelenthető, hogy az öröklődésnek a döntéskiemelés és döntésösszevonás szabályai szerint való alkalmazása a Liskov féle helyettesíthetőségi elv teljesülését eredményezi.

Nyitottság – zártság elve („The Open-Closed Principle”):
Rendszerünknek kiterjeszthetőség – bővíthetőség tekintetében nyitottnak kell lennie, módosíthatóság szerint zártnak.
A döntésszerkezet szempontjából a kiterjesztésre nyitottság szabálya akkor teljesül, ha a döntésredundanciákat döntéskiemeléssel és összevonással megszüntetjük, azaz a döntésszerkezet optimalizált. Optimális döntésstruktúra esetén egy új döntéslehetőség bevezetése könnyen megoldható, hiszen a döntéseseteknél, azaz a döntésfelhasználások helyén nem szükséges a programmódosítás. Új döntéslehetőség bevezetése csak a döntést definiáló osztályhierarchiában okoz változást, ahol egy új alosztályt kell bevezetni, ami a döntéslehetőség adat és viselkedés struktúráját megadja. Az első döntésesetnél fel kell készíteni a programot az új döntéslehetőség kezelésére, ami után a már archivált, bezárt döntés felhasználási helyein nem szükséges további módosítás.
Eszerint tehát a rendszerünk kiterjesztése rugalmasan megtehető, azaz nyitott.
A megfelelő döntésszerkezet alkalmazásával tehát lehetőség van az új viselkedés szerinti kiterjesztések elvégzésére, és eszerint a korábbi viselkedések működésének módosítása elkerülhető.
Az elv teljesülését segítik elő a „Sablonfüggvény” és „Stratégia” tervezési minták. A „Sablonfüggvény” tervezési minta ekvivalens döntéspredikátumú, de viselkedésében nem ekvivalens döntések összevonására ad megoldást, ahol az összevont döntések döntéslehetőségeinek kiterjesztése a felhasználási helyeken nem igényel

programmódosítást. A „Stratégia” tervezési minta ekvivalens viselkedési szerződésű döntések összevonását jellemzi, ahol nem követelmény a döntéspredikátumok ekvivalenciája. Kiterjesztés esetén nem szükséges az összevonás előtti döntések szerint több helyen elvégezni a döntéslehetőség hozzáadást.
Ugyanakkor az optimális döntésstruktúra a módosíthatóság ellen nem véd.

Függőség inverzió elve („Dependency Inversion Principle”):
Magasabb szintű modulok ne függjenek alacsonyabb szintű moduloktól, mindig absztrakciótól függjünk.
Döntésszerkezet szerint, ha a döntés már kiemelt állapotban deklarált, akkor egy absztrakt felületen keresztül érhető el, azaz a döntést egy polimorf metódus reprezentálja a felhasználás helyén. A döntéskiemeléssel, döntésredundanciák megszüntetésével, a döntések osztályszerkezetben történő ábrázolásával az absztrakciótól való függés elve megvalósul.

Vezérlési inverzió elve („Inversion Of Control”, „Hollywood Principle”):
„Ne hívj, majd mi hívunk”: A vezérlés a „központ” kezében van, amely azt megfelelő komponensek felé delegálja.
Az elv a döntés szerkezete szerint azt jelenti, hogy a döntéskiemelések után a döntésesetek kezelését központilag kell megoldani, ahol az egyes alsó szintű komponens hívások (döntéslehetőségeket reprezentáló alosztályok) mindig valamilyen absztrakt felületen keresztül történnek a „Függőség inverzió elvének” megfelelően.

A döntés és döntéskiemelés, a döntésredundanciák döntésösszevonással történő feloldásának bevezetése után az előzőekben ismertettem, hogy azok a már meglévő objektumorientált tervezési elvekhez miként viszonyulnak. A következőkben olyan új tervezési elvek bevezetését javaslom, amelyek a „Objektum összetétel használata öröklés helyett ahol lehet” elvvel ellentétben egyértelműen rávilágítanak arra, hogy az öröklés használata mikor indokolt: „Öröklést döntésredundancia megszüntetésre alkalmazzunk”. Továbbá egy absztraktabb elvet is megfogalmazok, amely nem csak az öröklés alkalmazásának témakörét tisztázza, hanem általános programszervezési elvként segítheti a szakembereket az optimálisabb megoldások keresésében.

Öröklést döntésredundanciák megszüntetésre alkalmazzunk:
A döntés, döntéskiemelés és döntésösszevonás definíciójából az következik, hogy az öröklés célja a döntések absztrakt definiálásával a döntésösszevonás lehetőségének megteremtése.
Kijelenthető, hogy az öröklés alkalmazása akkor jogos, ha a döntésszerkezeti függőségek azt indokolják, azaz, ha a döntésredundanciák valamelyik esete teljesül („3.2 A döntésredundanciák elkerülésének esetei”), ami a döntésösszevonást indokolttá teheti („3.3 Döntésösszevonás”).
Az elv teljesülésének mérésére a „3.7 Döntésszerkezet elemzése új metrikákkal” fejezetben a „**Döntéskiemeléssel létrejött öröklődések aránya**” metrikát vezetem be.

A döntésredundanciát kerüljük:
A „3.2 A döntésredundanciák elkerülésének esetei” fejezetben ismertetem a döntésredundanciák elkerülésének szabályait, illetve az elkerülés szabályainak eseteiből kiindulva bemutatom azokat a helyzeteket, amikor a döntésösszevonás és kiemelés indokolt lehet.
A javasolt tervezési elv szerint, ahol szükséges a döntésösszevonás, vagyis azt a redundáns döntések indokolják, a döntések összevonásával azokat szüntessük meg.
Az elv alkalmazásával döntésredundanciák elkerülésének szabályai szerint döntésszerkezetileg egy olyan optimális felépítéshez jutunk, amely kihat az előzőleg taglalt alapelvekre, mivel a döntésszerkezet minősége és az alapelvek teljesülése összefügg.
Az elv teljesülésének mérésére a „3.7 Döntésszerkezet elemzése új metrikákkal” fejezetben az **„Ekvivalens döntésesetek aránya”, „Döntéspredikátumában eltérő egyéb viselkedésében egyező döntésesetek aránya”, „Ekvivalens döntéspredikátumú döntésesetek aránya”** metrikákat vezetem be.

3.7 Döntésszerkezet elemzése új metrikákkal

A döntésredundancia megszüntetésére és elkerülésére bevezetett objektumorientált programozási alapszabályok elveivel párhuzamosan olyan új objektumorientált metrikák bevezetését javaslom, melyek a ciklikus komplexitás [16][41] meghatározásnál is figyelembe vett döntésszámon alapszanak, figyelembe véve a Qi[4] által is elemzett polimorf elágazásokat.
A javasolt metrikák közül az első kettő olyan metrika, amely meghatározásához nem szükséges olyan specifikációs nyelv alkalmazása, amellyel a programok viselkedési szerződései meghatározhatók.

Döntésabsztrakciós metrika:
Az új metrikát „Döntésabsztrakciós” metrikának nevezem, amellyel szemléltethető, hogy a döntésesetek és a polimorf döntésesetek milyen arányban állnak egymással:

polimorf döntésesetek száma/döntésesetek száma

Polimorf döntéseseteken azt értem, mikor döntéskiemelés után a döntést már egy osztályhierarchiában osztállyal és annak alosztályaival valósítom meg, és annak felhasználási helyein a szülőosztály típusú referencián keresztül hívjuk a döntést reprezentáló polimorf metódust. Minden egyes ilyen metódushívás egy döntés esetnek számít.
Döntéseseteken a programban lévő feltételes utasítások számának és a polimorf döntésesetek számának összegét értem.
Látni kell, hogy nem vizsgáljuk, hogy a megmaradt feltételes utasításokkal reprezentált döntések közül mely esetekben lenne még indokolt a kiemelés. Ennek eldöntéséhez a döntések viselkedési szerződésének vizsgálata szükséges.
Értéke 0-1 közötti lehet, ahol a magasabb érték jelzi, hogy a kód jól strukturált.
Feltételezésem szerint a polimorf döntések arányának növelése csökkenti a kód komplexitását és hozzájárul a karbantarthatóság növeléséhez.

Ez az új metrika a belső komplexitás mérésére alkalmas, azaz méret metrikák csoportjába tartozik.

Döntéskiemeléssel létrejött öröklődések aránya:
Az „**Öröklést döntésredundanciák megszüntetésre alkalmazzunk”** javasolt új tervezési alapelv teljesülésének méréséhez meg kell tudnunk állapítani, hogy egy öröklődéses osztályszerkezet esetén a megadott feltétel teljesül-e, vagyis az öröklődés alkalmazása döntésredundanciák megszüntetése érdekében történt-e.
Feltételezem, hogy, ha egy osztály-alosztály kapcsolatban van polimorf metódus (a kiemelt döntések felületeit a polimorf metódusok adják), akkor az adott öröklődés döntésredundanciák megszüntetésére jött létre. Tehát az elv teljesülésének megállapítása úgy lehetséges, hogy azt kell mérnünk, hogy az öröklődéses kapcsolatoknál van-e megadva polimorf metódus, amely az osztálystruktúrába kiemelt döntések felületét adja.
Az elv akkor teljesül nagyobb mértékben, ha a vizsgált osztályszerkezetekben lévő öröklődések nagyobb részben tartalmaznak polimorf metódusokat
A feltételezés az, hogy a kód integritása, szervezettsége annál nagyobb, minél nagyobb ez az arányszám.
Arányszám meghatározása:

polimorf metódusokat tartalmazó öröklődések száma/összes öröklődés száma

Értéke 0-1 közötti lehet, ahol a magasabb érték jelzi, hogy a kód jól strukturált.
Feltételezésem szerint az arányszám növelése csökkenti a kód komplexitását és hozzájárul a karbantarthatóság növeléséhez.

A következő mérőszámok meghatározásához a döntés definíciónál megadott döntéslehetőség viselkedési szerződések elemzése szükséges.
Az előző fejezetekben megadtam a döntésredundanciák típusait, amely esetekben a döntésösszevonás indokolt. Ezen döntésredundancia esetek mérésére további metrikák bevezetését javaslom, amelyek az előbb megadott „Döntésabsztrakciós” metrikával megadott mérőszámmal mutatott eredménynél szofisztikáltabban fejezik ki, hogy a döntésredundancia mely esetei rontják a program minőségét.
A döntésredundancia elkerülésének esetei szerint tehát a következő új eset-specifikus mérőszámokat vezethetjük be, melyek egyben „A döntésredundanciát kerüljük” tervezési alapelv teljesülésének mérésére is használhatók.

- A döntéseknek megfelelő viselkedés (a döntéslehetőségekben megadott működési logika- utasítássor, kiemelt döntés esetén metódusok) és állapotleírás/adatszerkezet csak egy helyen legyen definiálva, azaz ne legyen a döntéslehetőségeknek megfelelő viselkedés (döntéspredikátumok is ekvivalensek a viselkedés részeként) vagy/és állapotleírás, adatszerkezet többször megadva.

  „Ekvivalens döntésesetek aránya” metrika a programban lévő ekvivalens döntésesetek számának az összes döntéseset számához viszonyított arányát adja meg. Értéke 0-1 tartományban mozoghat. Magas értéke rossz struktúrájú programot jelez.

- Nem egyező döntéspredikátumú, de egyéb tulajdonságaiban egyező döntések és annak körülményei a programban ne szerepeljenek többször, kivéve, ha az adott eset nem teszi lehetővé a döntések összevonását.

  „Döntéspredikátumában eltérő egyéb viselkedésében egyező döntésesetek aránya” metrika a programban lévő azon döntésesetek számának arányát mutatják, melyek döntéspredikátumai nem ekvivalensek, de viselkedési szerződésük egyéb vonatkozásában igen. Értéke 0-1 tartományban mozoghat. Magas értéke rossz struktúrájú programot jelez.
- Ekvivalens döntéspredikátumú, de döntéslehetőségeik definícióiban eltérő döntésesetek ne ismétlődjenek.

  „Ekvivalens döntéspredikátumú döntésesetek aránya” metrika: A programban lévő ekvivalens döntéspredikátumú döntésesetek számának az összes döntéseset számához viszonyított arányát adja meg. Értéke 0-1 tartományban mozoghat. Magas értéke rossz struktúrájú programot jelez.

Látnunk kell, hogy ezen új metrikák meghatározásához a döntéslehetőségek viselkedési szerződéseinek elemzése szükséges. Lényeges tehát valamilyen specifikációs nyelv alkalmazása, mellyel a program specifikációs állításokra történő leképzése megtörténhet.

3.7.1 Döntésszerkezeti metrikák felhasználásának mérési eredményei

A döntésszerkezet elemzésével összefüggésben megfogalmazott metrikák kódintegritással kapcsolatos összefüggéseinek igazolására a WIT-SYS Consulting Zrt.[1] verziókövető adatbázisában tárolt Java alapú projektek forráskódjának elemzését végeztem el a projektek életciklusát lefedő szórással véve mintákat.
Mivel a mért projektek projektvezetője és vezető tervezője vagyok, ezért a mérési eredmények szerinti metrika érték alakulások és különböző „projekt események” között összefüggéseket vettem észre, melyeket szintén részletezek.
A fejlesztések során a projektek által alkalmazott minőségbiztosítási módszerek alapján a forráskódok állapota javult.
A fejlesztés minőségbiztosításának alapját a PMD[2], FindBugs[3], Checkstyle[4] kódminőség mérő eszközök folyamatos használata adta, amit manuális és automatikus tesztelési módszerek egészítettek ki.
Ennek megfelelően a forrásállapotokra alkalmazott metrikák értékének a strukturáltság javulását jelezve növekednie kell.

---

[1] http://www.wit-sys.hu
[2] http://pmd.sourceforge.net
[3] http://findbugs.sourceforge.net
[4] http://checkstyle.sourceforge.net

A méréshez olyan az Eszterházy Károly Főiskola által készített statikus kódelemzőt használtam, amellyel a forráskód következő paraméterei mérhetők:

- Öröklődések száma.
- Polimorf metódusokat tartalmazó öröklődések száma.
- „if-else” elágazások száma.
- Polimorf metódushívások száma.

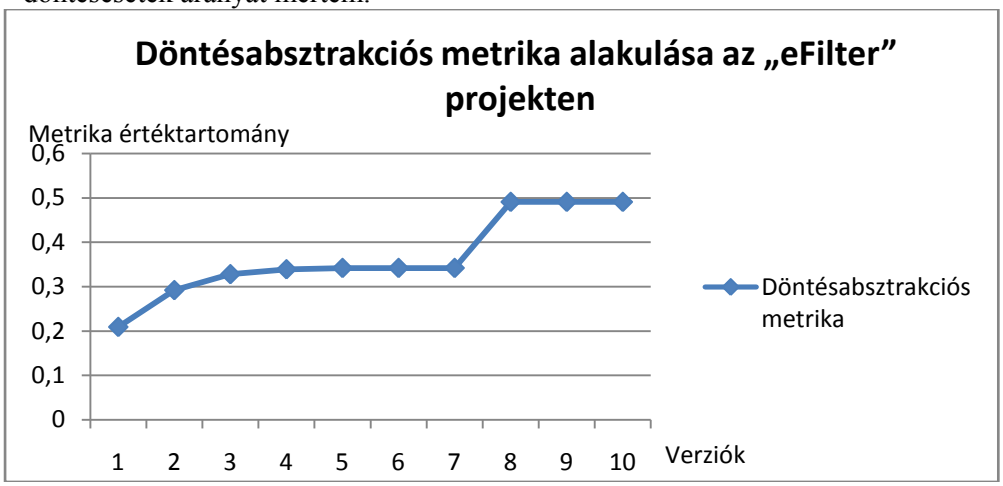
Az eszköz által mérhető paraméterek lehetőséget adtak arra, hogy a döntésszerkezet minőségének mérésére bevezetett metrikák egy részének az értékeit meghatározzam. A mérések eredményeit a következő alfejezetekben ismertetem.

3.7.1.1 Döntésabsztrakciós metrika mérési eredményei

A „**Döntésabsztrakciós metrika**” méréséhez szükséges meghatározni a következő arány értékét:

Polimorf döntésesetek száma/döntésesetek száma

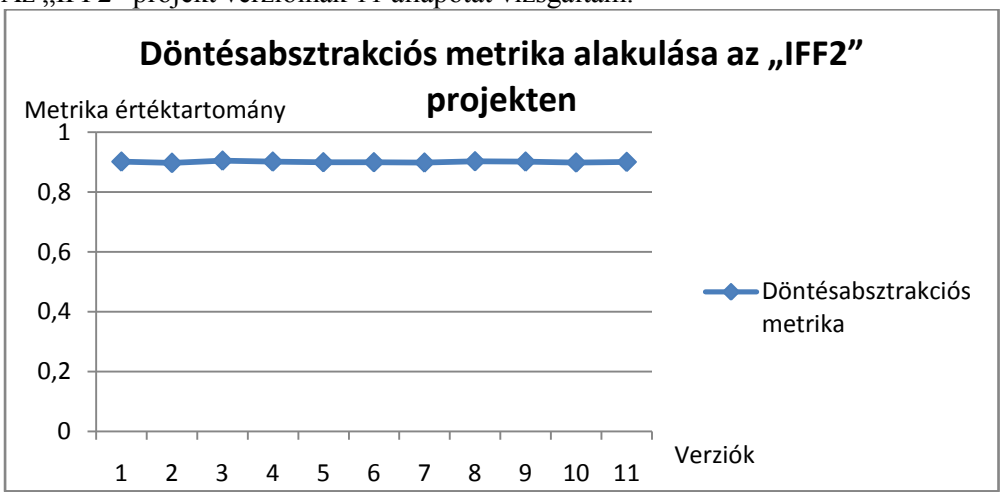
A vizsgálat során az „eFilter” kutatásfejlesztési projekt forrásállapotaiból 10 darab mintát vettem, melyeken a fentebb megadott képlet szerint a polimorf döntésesetek – döntésesetek arányát mértem.



6. Diagram: „Döntésabsztrakciós metrika” mérési eredményei az „eFilter” projekten

A vizsgálat eredménye jól mutatja, hogy a „Döntésabsztrakciós metrika” arányszáma a projekt-ciklus alatt hogyan növekedett. Ez igazolja azt a feltevést, miszerint a „Döntésabsztrakciós metrika” által mért polimorf döntésesetek arányának növekedése a kódminőség javulását eredményezi. A diagramon jól látható a 7.-9. verzió közötti „kiugrás”, amikor a PMD, FindBugs, Checkstyle (kódminőség mérő eszközök) segítségével egy refaktorálási folyamat kezdődött a projekten.

Az „IFF2” projekt verzióinak 11 állapotát vizsgáltam:



7. Diagram: „Döntésabsztrakciós metrika” mérési eredményei az „IFF2” projekten

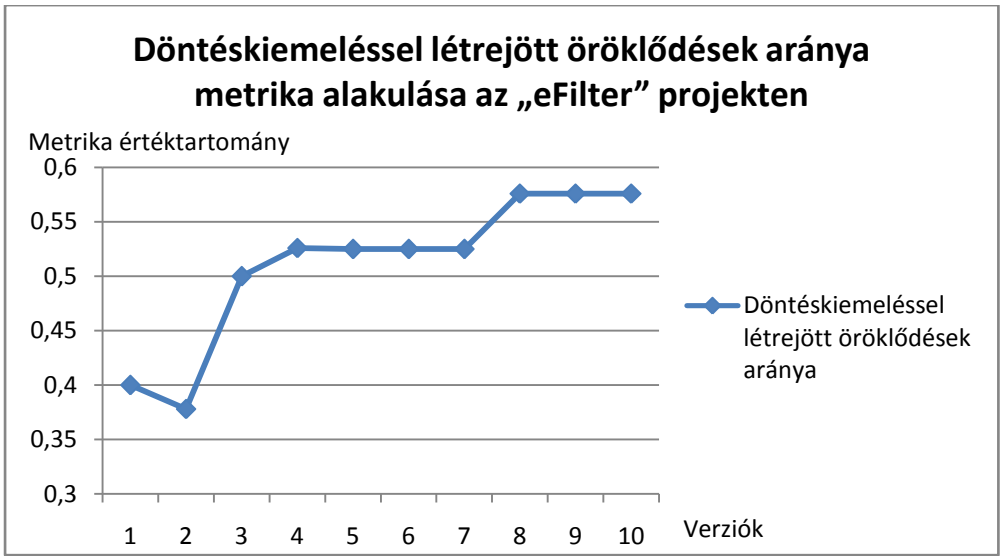
A „Döntésabsztrakciós metrika” értékei a mérési eredmények szerint az „IFF2” projekten torzítottak, mivel olyan tartományban mozognak, amit a projektre jellemző kódminőség nem indokol.
A „torzítást” az okozza, hogy az „IFF2” projekt üzleti logikájának, komplexebb algoritmusainak megvalósításai adatbázis oldalon találhatók, nem a vizsgált Java forráskódban. Ugyanakkor a felületkezelési logika kiépítése során a használt keretrendszer („JBoss Seam”) jellegéből fakadóan a polimorf metódushívások aránya jelentős, ami eltorzítja a metrika arányszámait.
Kijelenthető, hogy a metrikát csak komplexebb logikákat tartalmazó kód mérésére érdemes felhasználni.

3.7.1.2 Döntéskiemeléssel létrejött öröklődések aránya metrika mérési eredményei

A vizsgálat során a forrásállapotokból vett mintákon a következőket kell meghatározni:

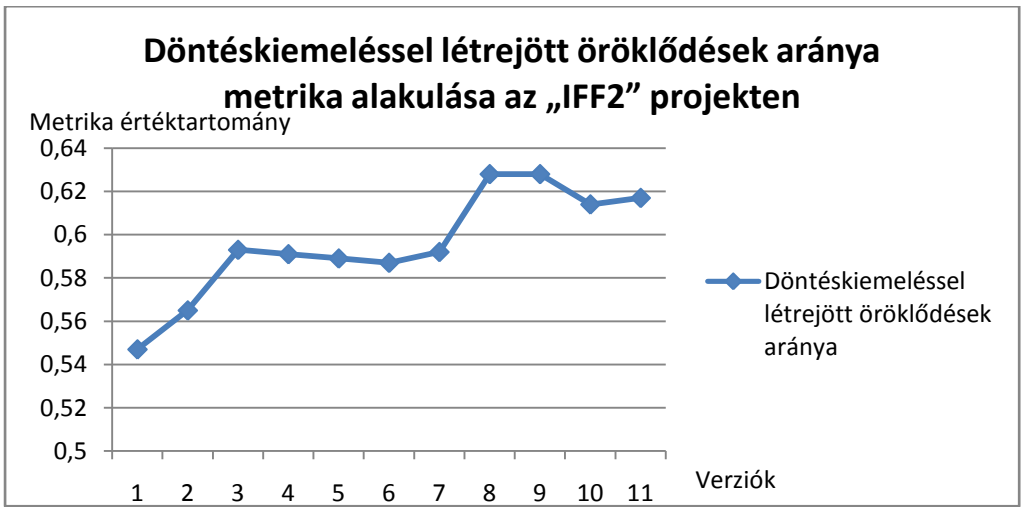
polimorf metódusokat tartalmazó öröklődések száma/összes öröklődés száma

Az „eFilter” kutatásfejlesztési projekt forrásállapotaiból 10 darab mintát vettem, amelyeken „Döntéskiemeléssel létrejött öröklődések aránya metrika” értékeit mértem:



8. Diagram: „Döntéskiemeléssel létrejött öröklődések aránya metrika” mérési eredményei az „eFilter” projekten

Az „eFilter” projekt forrásmintán végzett mérés jól mutatja, hogy a kezdeti fejlesztések után végrehajtott stabilizációt, illetve a 7.-9. tartó refaktorálási folyamat által megvalósuló minőségjavulást. Ez alapján kijelenthető, hogy a „Döntéskiemeléssel létrejött döntések aránya metrika” a kód minőségét jól szemlélteti.



9. Diagram: „Döntéskiemeléssel létrejött öröklődések aránya metrika” mérési eredményei az „IFF2” projekten

Annak ellenére, hogy az „IFF2” projekt komplexebb logikáinak megvalósítását nem a Java források tartalmazzák, a polimorfizmust kihasználó öröklődések arányának növekedése tapasztalható a projekt folyamán megvalósuló stabilizáció eredményeként. Érdekesség a 8.-10. verzió közötti „visszaesés”, amikor egy részprojekt keretében egy megrendelői kérés csomag került megvalósításra, időlegesen csökkentve a kódszerkezet minőségét.

4 Döntések vizsgálata JML-lel

Az objektumorientált programokban lévő döntések formális leírásánál szükséges, hogy legyen lehetőség mind az adatstruktúra mind a viselkedés formalizálására, hogy a döntéslehetőségek adatstruktúrája, illetve viselkedése összehasonlítható legyen. Meg kell tudni különböztetni a nem kiemelt, illetve a kiemelt állapotú döntéseket, illetve azok formális leírását. A döntésredundancia vizsgálat szempontjából lényeges, hogy a döntések döntés predikátumai [30] elemezhetőek legyenek.
Mivel a JML mind az adatstruktúra, mind a viselkedés specifikálására ad eszközöket, és az egyes metódusok viselkedését logikai formulákkal, mint utófeltételekkel valósítja meg, lehetőség van arra, hogy a döntések döntéslehetőségeinek egyezőségét mind az érintett adatstruktúra, mind a logikai állításokkal, utófeltételként megfogalmazott viselkedés alapján vizsgáljuk. Ugyanakkor, mivel mind az adatstruktúrát, mind a viselkedést logikai állításokkal fogalmazza meg, a döntés-predikátumok vizsgálata is megoldható.[31]

A fejezetben taglalt témakörök szerint új eredménynek a következőket tartom:
Döntéskiemelés során a döntésszerkezetek JML specifikációja miként változik:
A döntésszerkezetek JML alapú formalizálásával kapcsolatban új eredménynek tartom, az egyszintű és többszintű döntésszerkezetek JML alapú formalizáció módjának ismertetése során a nem kiemelt és az osztályszerkezetbe kiemelt döntések JML specifikációs lehetőségeinek bemutatását, amely alapján érzékelhető, hogy milyen változások következnek be a viselkedési szerződésben a kiemelés során.
4.2 Döntések formalizálása nem kiemelt állapotban
4.3 Döntések formalizálása kiemelt állapotban

Döntésösszevonás eseteinek JML alapú megfogalmazásai:
A „3.3 Döntésösszevonás” fejezetben ismertetettem azon eseteket, amelyekben a döntésredundancia megszüntetése indokolja a döntésösszevonást. Az ott megfogalmazott kritériumokat JML specifikációs eszközökkel is definiálom a „4.4 Döntésösszevonás eseteinek vizsgálata JML-lel” fejezetben.

A JML specifikációs eszközeinek rövid áttekintése után első lépésként bemutatom, hogy a döntéseket a JML-lel miként formalizálhatjuk, külön részletezve az egy szintű döntések, illetve a két/többszintű döntések formalizálásának módját.[31]
Hogy a döntések két állapota közötti összefüggéseket jobban megértsük, fontos bemutatni, hogy miként jelenik meg, ha a döntés még nincs kiemelve, illetve, ha már kiemelt formában van. Látható lesz tehát, hogy a döntések két állapota JML-lel formalizáltan miként adható meg. A kiemelt döntések formalizálásánál bemutatjuk, hogy miként záródik, archiválódik és kerül felhasználásra az adott döntés első döntésesete a későbbi felhasználások során (adott döntés későbbi döntésesetei).
Külön kitérek a döntésösszevonás eseteinek bemutatására, ahol a jól strukturáltság szabályainak teljesülése érdekében indokolt döntéskiemelések és összevonások eseteit a JML fogalomrendszerével részletezem.

Zárásul a nem kiemelt és kiemelt döntés JML-es formalizációs lehetőségeit egy példán keresztül mutatom be.

4.1 JML specifikációs eszközeinek rövid bemutatása

A következőkben vázlatosan bemutatom a JML specifikációs eszközeit, hogy a JML-ről egy átfogó képet kapva könnyebben tudjuk értelmezni az objektumorientált módszertan döntésalapú megközelítésének JML-lel történő elemzését.
A JML specifikációs eszközöket két csoportra oszthatjuk:

- A viselkedés specifikáció eszközei: Pl.: assert, assume, require, …
- Osztályok, interfészek specifikálása: Pl.: invariant, model, …

A viselkedés specifikáció eszközei:
Alapfogalom az utasítás, metódus elő és utófeltétele, amelyek meghatározzák a program állapotát egy utasítás, metódus végrehajtása előtt illetve után, amellyel egyben deklaráljuk az állapotátmenetet.
Az elő- és utófeltételeket következőképpen értelmezhetjük: [43]

- Előfeltétel:
    - A metódus, utasítás feltételezi az előfeltétel teljesülését.
    - A metódus, utasítás hívója biztosítja, hogy az előfeltétel teljesül.
- Utófeltétel:
    - A metódus, utasítás biztosítja az utófeltétel teljesülését.
    - A metódus, utasítás hívója feltételezi, hogy az utófeltétel teljesül.

Elő és utófeltételek a következőképpen jelennek meg a JML-ben:[43]

- Feltételek metóduson belül– Assertion
    - Assume: Állítás, amit a program megkövetel.
    - Assert: Állítás, amit a program biztosít.
- Feltételek metódusok között
    - Requires: Specifikálja a metódus előfeltételét.
    - Ensures: Specifikálja a metódus utófeltételét.

Van lehetőség a változók elő- és utóállapotainak megkülönböztetésére a következők szerint:

- Kezdő állapot: A változók kezdő állapotára, azaz pl. a metódus meghívása előtti állapotban érvényes változó értékekre úgy hivatkozunk, hogy a változó nevét \old() kifejezésbe zárjuk.
- Végállapotnak megfelelő változóértékekre egyszerűen a változónevekkel tudunk hivatkozni.

A metódusok specifikációjánál az assignable-lel bevezetve kell felsorolni azon változókat (keretfeltételek), melyek értékein az adott metódus módosíthat, vagyis ad keretet arra, hogy az adott metódus milyen állapotátmeneteket eredményezhet.
Érdemes megjegyezni, hogy a JML viselkedés alapú specifikációs eszközeivel megfelel a Hoare kalkulus elvárásainak [43].

Osztályok interfészek specifikálása:
A JML lehetőséget ad arra, hogy az osztályok, interfészek állapothalmazára általános érvényű feltételeket szabjunk meg – invariánsok. („invariant”)

A változókhoz lehetőség van „initially” kulcsszóval inicializáló feltételeket rendelni.
Érdekes specifikációs eszköz a „constraint” kulcsszóval megadható történeti megszorítás („history constraint”), ami a kezdő és végállapotok közötti összefüggésekre ad megkötéseket, amelyet minden az adott osztályt érintő állapotátmenetnek teljesíteni kell.
A JML lehetővé teszi, hogy változókat adatcsoportokba rendezzünk, amely struktúrák az állapot, állapotváltozás szempontjából egységként kezelendők.
Lehetőség van arra, hogy „model” kulcsszóval bevezetve modell szintű változókat definiáljunk, amelyeket csak a specifikációs elemek használhatnak. A modell szintű változók implementációs reprezentációit is specifikálni kell, amit csak azon a szinten kell megtennünk, ahol az adott követelmény megvalósítása ténylegesen megtörténik.

4.2 Döntések formalizálása nem kiemelt állapotban

Egy szintű döntés formalizálása
Ha a döntés nem kiemelt, akkor a döntéslehetőségeinek metodológiája és adatstruktúrája egy metódusban kerül megfogalmazásra, azaz nem egy osztály alosztályaiban.
A döntés formalizálásakor a döntést tartalmazó metódus specifikációjában „also” kulcsszóval elválasztva megfogalmazzuk a két döntéslehetőséget azok elő- és utófeltételeivel, melyekkel megadjuk a döntéslehetőségek viselkedését, azaz annak „lenyomatát”, következményeit:

$/*@\ \ public\ normal\_behavior\ \ //D$

$@\ \ requires\ \ p_1;\ //D_{L_1}$

$@\ \ assignable\ v_1,\ldots,v_a,v_b,\ldots,v_c;$

$@\ \ ensures\ \ feltétel_1\ \&\&\ldots\&\&\ feltétel_i\ \&\&\ feltétel_j\ \&\&\ldots\&\&\ feltétel_k;$

$@\ also$

$@\ \ requires\ \ !(p_1);\ //D_{L_2}$

$@\ \ assignable\ v_1,\ldots,v_a,v_e,\ldots,v_f;$

$@\ \ ensures\ \ feltétel_1\ \&\&\ldots\&\&\ feltétel_i\ \&\&\ feltétel_l\ \&\&\ldots\&\&\ feltétel_m;$

$@*/$

A döntés két döntéslehetőségének specifikációját az *also* kulcsszó választja el.
A döntéslehetőségek előfeltételei: $p_1, !p_1$, melyek a korábban bevezetett döntéspredikátumként értelmezhetők. Az előfeltétel teljesülése esetén jut érvényre az adott döntéslehetőség által módosításra kerülő adatstruktúra *assignable* kulcsszóval bevezetve, illetve az általuk meghatározott utófeltétel, mint viselkedés az *ensures* kulcsszóval bevezetve.
A döntéslehetőségek által érintett adatstruktúrákat, illetve az általuk megfogalmazott viselkedést tekintve meg kell különböztetnünk közös, illetve döntéslehetőség specifikus részeket. Ez lényeges a döntés kiemelésekor, mivel ezek alapján határozható meg, hogy kiemeléskor mely adatstruktúra és viselkedés lesz

specifikálva a szülőosztályban, illetve mely adatstruktúra és viselkedés részek lesznek valamelyik alosztályban megadva.

Döntéslehetőségek által érintett adatstruktúra:

$v_1,\ldots,v_a$ - Változók, melyek állapotát mindkét döntéslehetőség befolyásolja.

$v_b,\ldots,v_c$ - A D_{L_1} döntéslehetőség által érintett változók.

$v_e,\ldots,v_f$ - A D_{L_2} döntéslehetőség által érintett változók.

Döntéslehetőségek által meghatározott viselkedés:

$feltétel_1 \&\&\ldots\&\& feltétel_i$ - Mindkét döntéslehetőség által tartalmazott közös viselkedés.

$feltétel_j \&\&\ldots\&\& feltétel_k$ - A D_{L_1} döntéslehetőség által specifikált viselkedés.

$feltétel_l \&\&\ldots\&\& feltétel_m$ - A D_{L_2} döntéslehetőség által specifikált viselkedés.

Döntés tartalmaz döntést formalizálása:

Mivel az összetett döntésszerkezetek vizsgálata érdekes terület, amely a tervezési minták vizsgálata szempontjából is kulcsfontosságú, ezért szükséges a „döntés tartalmaz döntést” szerkezetek formális elemzésének előkészítése.

$/*@\ public\ normal\_behavior\ //D_2\ in\ D_1$

$@\ requires\ p_1;\ //D_{1L_1}$

$@\ \{|$

$@\quad requires\ p_2;\ //D_{2L_1}$

$@\quad assignable\ v_1,\ldots,v_a,v_b,\ldots,v_c,\ //D_{1L_1}$

$@\qquad v_g,\ldots,v_h,v_i,\ldots,v_j;//D_{2L_1}$

$@\quad ensures\ feltétel_1 \&\&\ldots\&\& feltétel_i \&\& feltétel_j \&\&\ldots\&\& feltétel_k \&\& //D_{1L_1}$

$@\qquad feltétel_p \&\&\ldots\&\& feltétel_q \&\& feltétel_r \&\&\ldots\&\& feltétel_s;//D_{2L_1}$

$@\ also$

$@\quad requires\ !p_2;\ //D_{2L_2}$

$@\quad assignable\ v_1,\ldots,v_a,v_b,\ldots,v_c,//D_{1L_1}$

$@\qquad v_g,\ldots,v_h,v_k,\ldots,v_l;//D_{2L_2}$

$@\quad ensures\ feltétel_1 \&\&\ldots\&\& feltétel_i \&\& feltétel_j \&\&\ldots\&\& feltétel_k \&\& //D_{1L_1}$

$@\qquad feltétel_p \&\&\ldots\&\& feltétel_q \&\& feltétel_t \&\&\ldots\&\& feltétel_u;//D_{2L_2}$

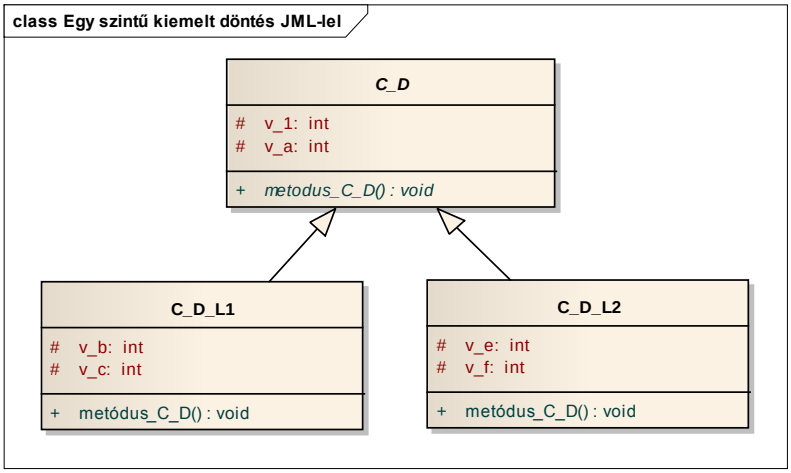
$@\ |\}$

```
@ also
@  requires  ! p1; //D1L2
@  {|
@    requires  p2; //D2L1
@    assignable v1,...,va,ve,...,vf; //D1L2
@                vg,...,vh,vi,...,vj; //D2L1
@    ensures  feltétel1 && ... && feltételi && feltétel l && ... && feltételm && //D1L2
@              feltételp && ... && feltételq && feltételr && ... && feltétels; //D2L1
@   also
@    requires ! p2; //D2L2
@    assignable v1,...,va,ve,...,vf; //D1L2
@                vg,...,vh,vk,...,vl; //D2L2
@     ensures  feltétel1 && ... && feltételi && feltétel l && ... && feltételm && //D1L2
@              feltételp && ... && feltételq && feltételt && ... && feltételu; //D2L2
@   |}
@*/
```

A D_1 döntés, illetve annak döntéslehetőségei (D_{1L_1}, D_{1L_2}) tartalmazzák a D_2 döntést illetve annak döntéslehetőségeit (D_{2L_1}, D_{2L_2}). A döntéslehetőségek specifikálása ugyanúgy történik, mint az egy szintű döntések esetén, azaz mind az érintett adatstruktúra, mind a specifikált viselkedés tekintetében vannak a döntéslehetőségek tekintetében közös, illetve döntéslehetőség-specifikus részek. Ezeket – az egy szintű döntésekhez hasonlóan – az adatstruktúra változók, illetve a viselkedés feltételeinek indexelésével jelezzük.
A döntéslehetőségeknek megfelelő viselkedést részben vagy egészben külön metódusba szervezheti a fejlesztő. Ebben az esetben az így kiemelt viselkedés követelményei a hívott metódus specifikációjába is kerülnek, de a fentebbiek szerint megmaradnak a kiemelés helyén is.

4.3 Döntések formalizálása kiemelt állapotban

Egy szintű kiemelt döntés formalizálása:



10. Diagram: Egy szintű döntés kiemelt formában: $C_D, C_{D_{L_1}}, C_{D_{L_2}}$

A döntések mentén az eredeti felépítés „széthasad” két alosztályra, kiemelve a szülőosztályba a közös részeket. Ezek a közös és döntéslehetőség-alosztály specifikus részek olvadnak össze egy működéssé a döntési helyeken.

A nem kiemelt döntés modell szintű JML specifikációja ugyanúgy megadható, mint a kiemelt döntés esetén. A különbség az, hogy a specifikáció megvalósítását milyen szinten realizáljuk, vagyis modellváltozók implementációs reprezentációja miként kerül megadásra. Erre nem kiemelt döntés esetén az adott osztályban, kiemelt döntés esetén az adott osztály valamelyik alosztályában kerül sor.
A szülőosztályban a döntés nem döntés specifikus részét megvalósításra alapozva adjuk meg. A döntés specifikus részekkel kapcsolatos követelmények megadása modellváltozók és azokra épülő viselkedési követelmények meghatározásával történik. A szülőosztályban a *D* döntés megjelenése a következő:

$/ * @\ public\ normal\_behavior\ //D_{L_1}$

$@\ \ requires\ modell\_p_1 ; // \Leftrightarrow p_1$

$@\ \ assignable\ v_1, v_a, modell\_v_b, modell\_v_c ;$

$@\ \ ensures\ \ feltétel_1\ \&\&\ldots\&\&\ feltétel_i\ \&\&\ modell\_feltétel_j\ \&\&\ldots\&\&\ modell\_feltétel_k ;$

$@\ also$

$@\ public\ normal\_behavior\ //D_{L_2}$

$@\ \ requires\ !modell\_p_1 ; // \Leftrightarrow !\ p_1$

$@\ \ assignable\ v_1, v_a, , modell\_v_e, modell\_v_f ;$

$@\ \ ensures\ \ feltétel_1\ \&\&\ldots\&\&\ feltétel_i\ \&\&\ modell\_feltétel_l\ \&\&\ldots\&\&\ modell\_feltétel_m ;$

$@ * /$

Döntéslehetőségek által érintett adatstruktúra:

$v_1,...,v_a$ - Változók, melyek állapotát mindkét döntéslehetőség befolyásolja.

$modell\_v_b,...,modell\_v_c$ - A D_{L_1} döntéslehetőség által érintett adatstruktúrák modell szintű reprezentánsai, amelyek implementációhoz kötése a D_{L_1} szerinti alosztályban történik.

$modell\_v_e,...,modell\_v_f$ - A D_{L_2} döntéslehetőség által érintett adatstruktúrák modell szintű reprezentánsai, amelyek implementációhoz kötése a D_{L_2} szerinti alosztályban történik.

Döntéslehetőségek által meghatározott viselkedés:

$feltétel_1 \&\&...\&\& feltétel_i$ - Mindkét döntéslehetőség által tartalmazott közös viselkedés.

$modell\_feltétel_j \&\&...\&\& modell\_feltétel_k$ - A D_{L_1} döntéslehetőség által specifikált viselkedés, amelyben modell szintű változók is szerepelnek.

$modell\_feltétel_l \&\&...\&\& modell\_feltétel_m$ - A D_{L_2} döntéslehetőség által specifikált viselkedés, amelyben modell szintű változók is szerepelnek.

Az alosztályokban a szülőosztályban megadott döntéslehetőség specifikus modell szintű követelményeket valósítjuk meg úgy, hogy a döntéslehetőségek viselkedése által érintett modell változókat konkrét implementációs változókhoz kötjük.

D_{L_1} döntéslehetőség a következő módon jelenik meg az egyik alosztályban:

```
/*@
@ public instance represents modell_p1 <- true;
@ public instance represents modell_vb <- vb;
@ public instance represents modell_vc <- vc;
*/

/*@ also //DL1
   @ public normal_behavior
   @  assignable vb,vc;
   @  ensures  feltétel_j && ... && feltétel_k;
@*/
```

D_{L_2} döntéslehetőség a következő módon jelenik meg a másik alosztályban:

```
/*@
@ public instance represents modell_p1 <- false;
@ public instance represents modell_ve <- ve;
@ public instance represents modell_vf <- vf;
*/
```

$/ * @ \ also \ //D_{L_2}$

$@ \ public \ normal\_behavior$

$@ \ \ assignable \ v_e, v_f;$

$@ \ \ ensures \ feltétel_1 \ \&\& \ldots \&\& \ feltétel_m;$

$@ * /$

A döntést reprezentáló polimorf metódus specifikációjában hivatkozott modell szintű változók implementációs szintű reprezentációt kapnak, amellyel a modell szintű változókat használó utófeltételek is implementációs függésbe kerülnek.
Ezek szerint a döntéslehetőségek által érintett implementációs szintű adatstruktúra:

$v_1, \ldots, v_a$ - Változok, melyek állapotát mindkét döntéslehetőség befolyásolja.

$v_b, \ldots, v_c$ - A D_{L_1} döntéslehetőség által érintett adatstruktúrák a D_{L_1} szerinti alosztályban megadott implementációs reprezentánsai.

$v_e, \ldots, v_f$ - A D_{L_2} döntéslehetőség által érintett adatstruktúrák a D_{L_2} szerinti alosztályban megadott implementációs reprezentánsai.

Döntéslehetőségek által meghatározott implementációs szintű viselkedés:

$feltétel_j \ \&\& \ldots \&\& \ feltétel_k$ - A D_{L_1} döntéslehetőség által specifikált viselkedés, amelyben modell szintű változókat már felváltották az implementációs szintű reprezentánsok.

$feltétel_l \ \&\& \ldots \&\& \ feltétel_m$ - A D_{L_2} döntéslehetőség által specifikált viselkedés, amelyben modell szintű változókat már felváltották az implementációs szintű reprezentánsok.

Ha a döntés már osztályhierarchiába kiemelten jelenik meg, akkor a döntés formalizálása a következők szerint változik:

- Döntéspredikátum: A döntéspredikátum modell szintű specifikálása a szülőosztályban van, amely szerint a szülőosztályban különválasztható a döntéslehetőségek szerinti viselkedés specifikációja. Implementációfüggő bekötése a döntéslehetőségek szerinti alosztályokban történik, a döntéslehetőségeknek megfelelő logikai literálhoz kötéssel. Értéket döntésarchiváláskor kap, amikor a megfelelő döntéslehetőség szerinti alosztály példányosításával a modell szintű predikátum implementációs szinten vagy az egyik döntéslehetőségnek megfelelő logikai igaz vagy a másik döntéslehetőségnek megfelelő logikai hamis értéket veszi fel.
- A döntés adatstruktúrái: Az adatstruktúra döntéslehetőségek szempontjából közös részének megadása az érintett változók szülőosztályban történő implementációs szintű deklarálásával történik. A döntéslehetőség specifikus adatszerkezetek esetén modell szintű változókat deklarálunk, amelyekhez az egyes alosztályok adják az implementációs szintű reprezentációt.
- A döntés viselkedése: A szülőosztályban megjelennek a döntéslehetőségek közös és döntéslehetőség specifikus részei is, amelyeket a döntéslehetőség specifikus részek esetén modell szintű változók felhasználásával specifikálunk. A döntéslehetőségeket reprezentáló alosztályok esetén a döntéslehetőség specifikus viselkedés megadása a szülőosztályban modell

szintű változók felhasználásával megadott viselkedésnek megfelelően történik úgy, hogy az ott felhasznált modell szintű változókat az adott alosztályban reprezentáló implementációs változókkal helyettesítjük.

Lehetőség van a döntéseseteknél, azaz a döntéshelyeken a kiemelést követően is az elvárt működést specifikálni egy „bezárt döntés” működésével kapcsolatosan. A következőkben *o* egy döntés szülőosztály típusú referenciával hivatkozott vagy a szülőosztály vagy döntéslehetőség szerinti alosztály példány:

- Döntéslehetőségek predikátumai: A döntés első előfordulásánál megmarad az eredeti $p_1, !p_1$ döntés predikátum. A későbbi előfordulási helyeken döntéspredikátum modell szintű deklarációjával helyettesítjük $o.modell\_p_1, !o.modell\_p_1$.
- A döntéslehetőségek által érintett adatstruktúráknak a szülőosztályban specifikált közös részeit, illetve az alosztályokban specifikált döntéslehetőség specifikus részeket egy szülőosztály típusú referenciával érhetjük el. Mivel a szülőosztályban a döntéslehetőség specifikus részek megadása is megtörténik modell szintű változókkal, ezért típuskényszerítésre nincs szükség az érintett adatstruktúrák megadásánál. Az adott *o* objektum implementációs és modell szintű változóit kell felsorolni az adott döntés által érintett változók listájában

$$D_{L_1} \Rightarrow assignable\ o.v_1, o.v_a, o.modell\_v_b, o.modell\_v_c;$$

$$D_{L_2} \Rightarrow assignable\ o.v_1, o.v_a, o.modell\_v_e, o.modell\_v_f;$$

- A döntés viselkedését az előző szabályok szerint elért adatstruktúrák segítségével deklarálhatjuk a szülőosztályban megadott döntéssel kapcsolatos viselkedéssel összhangban.

Ezek szerint a döntéskiemelés helyén vagy a későbbi döntésesetekben, ha a kiemelt döntéssel kapcsolatos követelményeket elő akarjuk írni, akkor a deklaráció módja a következő:

$/ * @\ public\ normal\_behavior\ //D_{L_1}$

$@\ \ requires\ o.modell\_p_1; //\Leftrightarrow p_1$

$@\ \ assignable\ o.v_1, o.v_a, o.modell\_v_b, o.modell\_v_c;$

$@\ \ ensures\ \ feltétel_1\ \&\&\ldots\&\&\ feltétel_i\ \&\&\ modell\_feltétel_j\ \&\&\ldots\&\&\ modell\_feltétel_k;$

$@\ also$

$@\ public\ normal\_behavior\ //D_{L_2}$

$@\ \ requires\ !o.modell\_p_1; //\Leftrightarrow !p_1$

$@\ \ assignable\ o.v_1, o.v_a, o.modell\_v_e, o.modell\_v_f;$

$@\ \ ensures\ \ feltétel_1\ \&\&\ldots\&\&\ feltétel_i\ \&\&\ modell\_feltétel_l\ \&\&\ldots\&\&\ modell\_feltétel_m;$

$@ * /$

A döntéskiemelés helyén az adott döntéssel kapcsolatos követelmény lenyomatát a kiemelt döntéshierarchia szülőosztályának adott döntést megvalósító polimorf metódusának hivatkozásával is megadhatjuk a következő módon:

$/*@$ *public normal_behavior //D*

$@$ *assignable /nothing;*

$@$ *ensures* $feltétel_1 \&\& \ldots \&\& feltétel_o \&\& o.polimorfMetódus();$

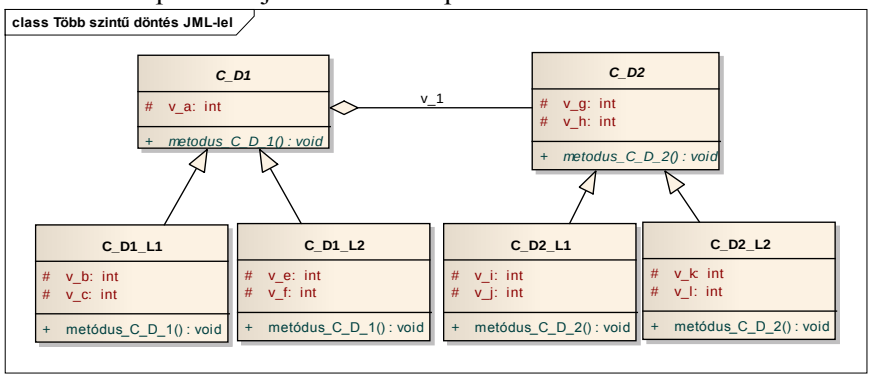
$@*/$

Ez csak abban az esetben tehető meg, ha a metódus „pure”, azaz mellékhatásmentes. Ha az adott polimorf metódusnak van mellékhatása és a kiemelt döntés követelményével kapcsolatban szeretnénk megkötést adni, akkor azt csak az előzőleg megadott módon tehetjük meg.

Döntés tartalmaz döntést kiemelt formában

Ha egy „döntés tartalmaz döntést” szerkezet már kiemelt állapotban van, akkor mindkét döntést egy osztállyal és annak alosztályaival adjuk meg, melyekre az egy szintű döntésszerkezetnél ismertetett szabályok teljesülnek.

Az egy szintű döntés esetén megadott szabályokat ki kell egészíteni azzal, hogy ebben az esetben az egyik döntésszerkezethez (tartalmazó döntés) tartozó osztályhierarchia polimorf metódusában felhasználjuk a másik döntéshez (tartalmazott döntés) tartozó osztályhierarchia polimorf metódusát. Ezek a polimorf metódusok reprezentálják kiemelt állapotban a két döntést.



11. Diagram: C_{D_1}, C_{D_2} osztályhierarchiák UML diagramja [40]

A következőkben megadjuk a kiemelt, összetett döntésszerkezetek JML-es formalizációs lehetőségeit.

Kiemelt tartalmazó döntés JML specifikációja:

A szülőosztályban a D_1 döntés és a tartalmazott D_2 döntés megjelenése a döntést megadó polimorf metódus specifikációjában a következő:

Abban az esetben, ha a tartalmazott döntéshierarchia polimorf metódusa nem „pure”, azaz nem mellékhatásmentes, akkor az azzal kapcsolatos követelményeket már ezen a szinten (tartalmazó döntés specifikációjának részeként) meg kell adnunk.

```
/*@ public normal_behavior //D2 in D1
  @ requires modell_p1; //D1L1
  @ {|
  @   requires modell_p2; //D2L1
  @   assignable v1,...,va,modell_vb,...,modell_vc, //D1L1
  @           v1.vg,...,v1.vh,v1.modell_vi,...,v1.modell_vj; //D2L1
  @   ensures feltétel1 && ... && feltételj && modell_feltételj && ... && modell_feltételk && //D1L1
  @           feltételp && ... && feltételq && modell_feltételr && ... && modell_feltétels; //D2L1
  @ also
  @   requires !modell_p2; //D2L2
  @   assignable v1,...,va,modell_vb,...,modell_vc, //D1L1
  @           v1.vg,...,v1.vh,v1.modell_vk,...,v1.modell_vl; //D2L2
  @   ensures feltétel1 && ... && feltételj && modell_feltételj && ... && modell_feltételk && //D1L1
  @           feltételp && ... && feltételq && modell_feltételt && ... && modell_feltételu; //D2L2
  @ |}
 @ also
  @ requires !modell_p1; //D1L2
  @ {|
  @   requires modell_p2; //D2L1
  @   assignable v1,...,va,modell_ve,...,modell_vf; //D1L2
  @           v1.vg,...,v1.vh,v1.modell_vi,...,v1.modell_vj; //D2L1
  @   ensures feltétel1 && ... && feltételj && modell_feltételj && ... && modell_feltételm && //D1L2
  @           feltételp && ... && feltételq && modell_feltételr && ... && modell_feltétels; //D2L1
  @ also
  @   requires !modell_p2; //D2L2
  @   assignable v1,...,va,modell_ve,...,modell_vf; //D1L2
  @           v1.vg,...,v1.vh,v1.modell_vk,...,v1.modell_vl; //D2L2
  @   ensures feltétel1 && ... && feltételj && modell_feltételj && ... && modell_feltételm && //D1L2
  @           feltételp && ... && feltételq && modell_feltételt && ... && modell_feltételu; //D2L2
  @ |}
@*/
```

A nem összetett döntés esetén leírtaknál taglaltak szerint a szülőosztályban a D_1 döntés nem döntés specifikus részét megvalósításra alapozva adjuk meg, a döntés specifikus részekkel kapcsolatos követelmények megadása modellváltozók, és azokra épülő viselkedési követelmények meghatározásával történik.

D_1 döntés által érintett adatstruktúra:

$v_1,\ldots,v_a$ - Változók, melyek állapotát mindkét döntéslehetőség befolyásolja.

$modell\_v_b,...,modell\_v_c$ - A D_{1L_1} döntéslehetőség által érintett adatstruktúrák modell szintű reprezentánsai, amelyek implementációhoz kötése a D_{1L_1} szerinti alosztályban történik.

$modell\_v_e,...,modell\_v_f$ - A D_{1L_2} döntéslehetőség által érintett adatstruktúrák modell szintű reprezentánsai, amelyek implementációhoz kötése a D_{1L_2} szerinti alosztályban történik.

D_1 döntés által meghatározott viselkedés:

$feltétel_1 \&\&...\&\& feltétel_i$ - Mindkét döntéslehetőség által tartalmazott közös viselkedés.

$modell\_feltétel_j \&\&...\&\& modell\_feltétel_k$ - A D_{1L_1} döntéslehetőség által specifikált viselkedés, amelyben modell szintű változók is szerepelnek.

$modell\_feltétel_l \&\&...\&\& modell\_feltétel_m$ - A D_{1L_2} döntéslehetőség által specifikált viselkedés, amelyben modell szintű változók is szerepelnek.

A D_2 döntéssel kapcsolatos specifikációkat a D_2 döntést reprezentáló v_1 referencián keresztül érjük el, ahol a D_2 döntés nem döntés specifikus részeit modell változókkal és azokra épülő viselkedési követelményekkel specifikáljuk.

D_2 döntés által érintett adatstruktúra:

$v_1.v_g,...,v_1.v_h$ - Változók, melyek állapotát mindkét döntéslehetőség befolyásolja.

$v_1.modell\_v_i,...,v_1.modell\_v_j$ - A D_{2L_1} döntéslehetőség által érintett adatstruktúrák modell szintű reprezentánsai, amelyek implementációhoz kötése a D_{2L_1} szerinti alosztályban történik.

$v_1.modell\_v_k,...,v_1.modell\_v_l$ - A D_{2L_2} döntéslehetőség által érintett adatstruktúrák modell szintű reprezentánsai, amelyek implementációhoz kötése a D_{2L_2} szerinti alosztályban történik.

D_2 döntés által meghatározott viselkedés:

$feltétel_p \&\&...\&\& feltétel_q$ - Mindkét döntéslehetőség által tartalmazott közös viselkedés.

$modell\_feltétel_r \&\&...\&\& modell\_feltétel_s$ - A D_{2L_1} döntéslehetőség által specifikált viselkedés, amelyben modell szintű változók is szerepelnek.

$modell\_feltétel_t \&\&...\&\& modell\_feltétel_u$ - A D_{2L_2} döntéslehetőség által specifikált viselkedés, amelyben modell szintű változók is szerepelnek.

A tartalmazó D_1 döntés döntéslehetőségek szerinti alosztályokban a szülőosztályban megadott döntéslehetőség specifikus modell szintű követelményeket valósítjuk meg úgy, hogy a döntéslehetőségek viselkedése által érintett modell változókat konkrét implementációs változókhoz kötjük.

D_{1L_1} döntéslehetőség a következő módon jelenik meg az egyik alosztály az adott döntést megvalósító polimorf metódus specifikációjában:

```
/*@
@ public instance represents modell_p1 <- true;
@ public instance represents modell_vb <- vb;
@ public instance represents modell_vc <- vc;
*/
/*@ also //D1L1
   @ public normal_behavior
   @  assignable vb,vc;
   @  ensures  feltétel_j && ... && feltétel_k;
@*/
```

D_{1L_2} döntéslehetőség a következő módon jelenik meg a másik alosztályban adott döntést megvalósító polimorf metódus specifikációjában:

```
/*@
@ public instance represents modell_p1 <- false;
@ public instance represents modell_ve <- ve;
@ public instance represents modell_vf <- vf;
*/
/*@ also //D1L1
   @ public normal_behavior
   @  assignable ve,vf;
   @  ensures  feltétel_l && ... && feltétel_m;
@*/
```

Ha a tartalmazott döntés polimorf metódusa mellékhatásmentes, azaz „*pure*”, akkor lehetőség van arra, hogy a tartalmazott döntés a következő módon jelenjen meg:

```
/*@  public normal_behavior //D1
   @  assignable va;
   @  ensures  feltétel_1 && ... && feltétel_o && v1.polimorfMetódus();
@*/
```

A tartalmazott döntés polimorf metódusát egyszerűen „meghivatkozzuk” az utófeltételek között. Erre csak „*pure*” metódus esetén van lehetőség.

Kiemelt tartalmazott döntés JML specifikációja:
Tartalmazott döntés polimorf metódusának JML-es specifikációja a kiemelt tartalmazott döntéshez tartozó osztályhierarchia szülőosztályában a következők szerint történik:

$/ * @\ public\ normal\_behavior\ //D_{2L_1}$

$@\ \ requires\ modell\_p_2;\ //\Leftrightarrow p_2$

$@\ \ assignable\ v_g, v_h, modell\_v_i, modell\_v_j;$

$@\ \ ensures\ \ feltétel_p\ \&\&\ldots\&\&\ feltétel_q\ \&\&\ modell\_feltétel_r\ \&\&\ldots\&\&\ modell\_feltétel_s;$

$@\ also$

$@\ public\ normal\_behavior\ //D_{2L_2}$

$@\ \ requires\ !modell\_p_2;\ //\Leftrightarrow !\ p_2$

$@\ \ assignable\ v_g, v_h, modell\_v_k, modell\_v_l;$

$@\ \ ensures\ \ feltétel_p\ \&\&\ldots\&\&\ feltétel_q\ \&\&\ modell\_feltétel_t\ \&\&\ldots\&\&\ modell\_feltétel_u;$

$@ * /$

A nem összetett döntés esetén leírtaknál taglaltak szerint a szülőosztályban a D_2 döntés nem döntés specifikus részét megvalósításra alapozva adjuk meg, a döntés specifikus részekkel kapcsolatos követelmények megadása modellváltozók és azokra épülő viselkedési követelmények meghatározásával történik.

D_2 döntés által érintett adatstruktúra:

$v_g, \ldots, v_h$ - Változók, melyek állapotát mindkét döntéslehetőség befolyásolja.

$modell\_v_i, \ldots, modell\_v_j$ - A D_{2L_1} döntéslehetőség által érintett adatstruktúrák modell szintű reprezentánsai, amelyek implementációhoz kötése a D_{2L_1} szerinti alosztályban történik.

$modell\_v_k, \ldots, modell\_v_l$ - A D_{2L_2} döntéslehetőség által érintett adatstruktúrák modell szintű reprezentánsai, amelyek implementációhoz kötése a D_{2L_2} szerinti alosztályban történik.

D_2 döntés által meghatározott viselkedés:

$feltétel_p\ \&\&\ldots\&\&\ feltétel_q$ - Mindkét döntéslehetőség által tartalmazott közös viselkedés.

$modell\_\ feltétel_r\ \&\&\ldots\&\&\ modell\_\ feltétel_s$ - A D_{2L_1} döntéslehetőség által specifikált viselkedés, amelyben modell szintű változók is szerepelnek.

$modell\_\ feltétel_t\ \&\&\ldots\&\&\ modell\_\ feltétel_u$ - A D_{2L_2} döntéslehetőség által specifikált viselkedés, amelyben modell szintű változók is szerepelnek.

A tartalmazott D_2 döntés döntéslehetőségek szerinti alosztályokban a szülőosztályban megadott döntéslehetőség specifikus modell szintű követelményeket valósítjuk meg úgy, hogy a döntéslehetőségek viselkedése által érintett modell változókat konkrét implementációs változókhoz kötjük.

D_{2L_1} döntéslehetőség a következő módon jelenik meg az egyik alosztályban:

$/*@$
$@\ public\ instance\ represents\ modell\_p_2\ <-true;$
$@\ public\ instance\ represents\ modell\_v_i\ <-v_i;$
$@\ public\ instance\ represents\ modell\_v_j\ <-v_j;$
$*/$
$/*@\ also\ //\ D_{2L_1}$
$@\ public\ normal\_behavior$
$@\ \ assignable\ v_i, v_j;$
$@\ \ ensures\ \ feltétel_r\ \&\&\ldots\&\&\ feltétel_s;$
$@*/$

D_{2L_1} döntéslehetőség a következő módon jelenik meg a másik alosztályban:

$/*@$
$@\ public\ instance\ represents\ modell\_p_2\ <-\ false;$
$@\ public\ instance\ represents\ modell\_v_k\ <-v_k;$
$@\ public\ instance\ represents\ modell\_v_l\ <-v_l;$
$*/$
$/*@\ also\ //\ D_{2L_2}$
$@\ public\ normal\_behavior$
$@\ \ assignable\ v_k, v_l;$
$@\ \ ensures\ \ feltétel_t\ \&\&\ldots\&\&\ feltétel_u;$
$@*/$

Kiemelés helyén a tartalmazott döntés JML specifikációja:
A tartalmazott döntés megjelenései a kiemelés helyén a tartalmazó döntés specifikációjában szerepel, ahol egy objektum referencián keresztül érjük el a tartalmazott döntést reprezentáló objektumot.

4.4 Döntésösszevonás eseteinek vizsgálata JML-lel

A „3.2 A döntésredundanciák elkerülésének esetei” fejezetben megadtam, hogy a jól strukturáltság szabályai szerint milyen szabályok teljesítésével lehet a döntésredundanciát elkerülni, amely alapján részben választ kapunk arra is, hogy milyen esetekben indokolt a döntésösszevonás. Ezeket a feltételeket JML-es fogalomrendszerrel is megvizsgálom.
A következő esetekben indokolt a döntésösszevonás:

1. Ha a döntések döntéspredikátumai ekvivalensek, illetve döntéslehetőségei olyan adattartalmat és funkcionalistást deklarálnak, amelyek egyeznek, részben egyeznek, egyik a másik kiterjesztése, egy közös résznek mindkettő kiterjesztése, akkor összevonhatók.
   JML-es specifikáció szerint vizsgálva a döntéseket a következők teljesülése esetén szükséges a döntések kiemelés:

- Keret feltételek - Döntések által érintett adatstruktúrák halmazai („*assignable*” adatstruktúrák - L_{D_1}, L_{D_2}) részben vagy egészben egyeznek, azaz van metszetük. Adatszerkezet: A döntések által érintett adatszerkezetet, mint keret feltételeket vizsgáljuk. A döntések adatszerkezetei tekintetében a következő viszonyulásokat különböztethetjük meg:
  - A döntések által érintett *assignabe* kulcsszóval specifikált változók halmazai egyeznek: $L_{D_1} = L_{D_2}$
  - Az egyik döntés keretfeltételeit adó változók a másik döntés keret feltételeit adó változóknak részhalmazát képezik: $L_{D_1} \subset L_{D_2}$, azaz az általuk leírható állapotok halmazai ezzel egyező relációval szemléltethetők.
  - Az érintett döntés keretfeltételeit adó változóknak van metszetük, azaz egyik változó halmaz sem részhalmaza a másiknak. Ezen esetben egy döntés adatszerkezetét mindkét vizsgált döntés adatszerkezet kiterjeszti: $L_{D_1} \cap L_{D_2} = L_D$, ahol az L_D a két döntés adatszerkezetének a közös része.
- Viselkedés:
  - A „*requires*” kulcsszóval megadott feltételek ekvivalensek, azaz a döntések döntéspredikátumaira: $P_{D_1} = P_{D_2}$
  - A döntéslehetőségek utófeltételei
    $utófeltétel_{D_{1L_1}}, utófeltétel_{D_{1L_2}}, utófeltétel_{D_{2L_1}}, utófeltétel_{D_{2L_2}}$
    tekintetében a következő eseteket határozhatjuk meg:
    - Döntések döntéslehetőségeinek utófeltételei ekvivalensek:
      $utófeltétel_{D_{1L_1}} = utófeltétel_{D_{2L_1}}, utófeltétel_{D_{1L_2}} = utófeltétel_{D_{2L_2}}$
    - Az egyik döntés utófeltételeinek egy része ekvivalens a másik döntés döntéslehetőség utófeltételeivel, kiegészítve azt további megszorításokkal.
      $utófeltétel_{D_{2L_1}} = utófeltétel_{D_{1L_1}} \wedge utófeltétel\_kiegészítés_{D_{2L_1}}$
      $utófeltétel_{D_{2L_2}} = utófeltétel_{D_{1L_2}} \wedge utófeltétel\_kiegészítés_{D_{2L_2}}$
      ahol az egyik döntés utófeltételeinek
      $utófeltétel\_kiegészítés_{D_{2L_1}}, utófeltétel\_kiegészítés_{D_{2L_2}}$
      kiegészítéseivel lesznek ekvivalensek a másik döntés utófeltételei.
    - A döntések utófeltételeinek van egy ekvivalens része, amely mindkét döntés esetén kiegészül további megszorításokkal
      $utófeltétel_{D_{1L_1}} = utófeltétel_{D_{L_1}} \wedge utófeltétel\_kiegészítés_{D_{1L_1}}$
      $utófeltétel_{D_{1L_2}} = utófeltétel_{D_{L_2}} \wedge utófeltétel\_kiegészítés_{D_{1L_2}}$

$$utófeltétel_{D_{2L_1}} = utófeltétel_{D_{L_1}} \wedge utófeltétel\_kiegészítés_{D_{2L_1}}$$
$$utófeltétel_{D_{2L_2}} = utófeltétel_{D_{L_2}} \wedge utófeltétel\_kiegészítés_{D_{2L_2}}$$

Ezek szerint a D_1 döntés $D_{1_{L_1}}, D_{1_{L_2}}$ döntéslehetőségének utófeltételei a D_2-vel ekvivalens közös részeket

$$utófeltétel_{D_{L_1}}, utófeltétel_{D_{L_2}}$$

egészítik ki döntés specifikus utófeltételekkel:

$$utófeltétel\_kiegészítés_{D_{1L_1}}, utófeltétel\_kiegészítés_{D_{1L_2}}$$

A D_2 esetén ezzel egyező módon egészülnek ki az ekvivalens utófeltételek döntés specifikus utófeltételekkel. Az így meghatározott ekvivalens rész-utófeltételek a következők:

$$utófeltétel_{D_{L_1}}, utófeltétel_{D_{L_2}}$$

2. Ha a döntések döntéslehetőségei által megadott viselkedések döntéspredikátumok nélküli előfeltételei, utófeltételei egyeznek, vagy az egyik vagy mindkettő egy közös viselkedést deklaráló előfeltételhez, utófeltételhez képest ad meg további megszorításokat.

   Az előző esetnél leírtak teljesülnek kivéve azt, hogy a „*requires*” kulcsszóval megadott feltételek nem ekvivalensek, azaz a döntések döntéspredikátumaira: $P_{D_1} \neq P_{D_2}$

3. Ha viselkedésük elő- és utófeltételei közül csak az előfeltételek részét képező döntéspredikátumok egyeznek, amikor az adatszerkezet egyezőség sem feltétel.
   JML formalizáció szerint a döntések összevonhatók, ha a döntéslehetőségek adatstruktúrája és metodológiája az előző fejezetben leírtak szerint részben vagy teljesen nem egyezik, azonban a döntéslehetőségek *„requires”* kulcsszóval megadott előfeltétel predikátumai ekvivalensek. Ezek szerint a döntések kiemelésére akkor van szükség, ha azok JML specifikációjára a következők teljesülnek:
   - Keretfeltételek – adatszerkezet: Adatszerkezet, keretfeltételek tekintetében nincs megkötés.
   - Viselkedés:
     Előfeltételek: A vizsgált döntések JML specifikációjában azok előfeltételi „*requires*”-el megadott feltételek ekvivalensek, azaz a döntések döntéspredikátumaira: $P_{D_1} = P_{D_2}$
     Utófeltételek: Utófeltétek tekintetében sincs megkötés.

5 Tervezési minták döntésalapú vizsgálata

Az előző fejezetekben bemutatott objektumorientált fogalmak és a tervezési alapelvek értelmezésének kiterjesztésére alapozva a tervezési minták is új szemszögből közelíthetők meg. A döntésszerkezeteket vizsgálva a tervezési minták másképpen is jellemezhetők, mellyel vélhetően céljaik világosabbá válnak, illetve a felhasználhatóságuk is jelentősen növelhető.

Refaktorálás támakörében pár tervezési minta esetén már megfogalmazták, hogy a feltételes utasítások milyen eseteiben szükséges kód optimalizációs szempontból döntéskiemelés alkalmazásával valamely, pl. „Stratégia" tervezési minta szerinti osztály struktúra bevezetése, ahol a döntési ágak, döntéslehetőségek szerinti viselkedéseket külön-külön alosztályokba sorolják.[18]
A „Stratégia" tervezési mintán kívül még a „Parancs", „Állapot" tervezési minták esetén mutatják be, hogy milyen „if-else" szerkezetek esetén szükséges az adott minták alkalmazásával optimálisabb osztály szerkezet kialakítása.[18]
Ugyanakkor kijelenthető, hogy ezen tervezési mintákra vonatkozó döntés szerkezetek szerinti elemzések nem vizsgálják átfogóan a döntésszerkezetek és a tervezési minták kapcsolatát. Elképzelésem szerint az előbb említett példákon felül a tervezési minták jelentős részére kijelenthető, hogy közöttük és a különböző döntésszerkezetek között egyértelmű összefüggések vannak, melyek vizsgálata lényegesen javíthatja a tervezési minták bevezethetőségének, alkalmazásának esélyeit.
Ezen koncepció szerint a tervezési minták célja a jól strukturáltság elérése, amit a döntésfüggőségek, azaz döntésismétlődések megszüntetésével érhetünk el. A **tervezési minták a döntésredundanciák megszüntetésére adnak recepteket**, vagyis arra, hogy a különböző döntés struktúrák esetén a döntésismétlődés megszüntetése miként biztosítható. [30][31]
Ha döntésalapú megközelítés szerint vizsgáljuk a tervezési minták szerkezetét, akkor az elképzelés lehetőséget ad arra, hogy a tervezési mintákat nem csak a jelenleg ismert osztály és objektum szerkezetük alapján ismerjük fel, hanem a döntések egymásba épülése, döntésszerkezete szerint. Kijelenthető, hogy **a tervezési minták tervezési helyzet és nem osztályszerkezet függőek. A tervezési helyzeteket a döntési struktúrák konkrét tervezési szerkezettől függetlenül tudják ábrázolni.**

A tervezési minták osztályozhatók aszerint, hogy milyen döntésszerkezetek esetén adják meg az optimális döntés-deklarációkat, melyekkel a döntésredundancia elkerülése biztosítható. Két csoportra oszthatók: [30][31]

- Döntésarchiváló tervezési minták: Ha egy döntést meghozva az egyes döntési pontokon az adott döntésnek megfelelően, de új adattartalmat és metodológiát kell gyártani, akkor az adott döntésszerkezeteknek megfelelő döntésarchiváló tervezési mintákat kell használni. A termékek strukturáltsága határozza meg, hogy melyik tervezési mintát kell használni. Az adja meg a döntésarchiválás módját, hogy az archiválandó döntés

körülmény, azaz a döntésnek megfelelő metodológia és adatszerkezet milyen tulajdonságokkal rendelkezik.
- Döntésszerkezetek optimális struktúrájának kialakításának eseteit taglaló tervezési minták: Meg kell határoznunk a döntések egymásba épülésének módozatait, ami alapján el tudjuk dönteni, hogy egy adott döntésstruktúra esetén milyen módon, azaz melyik tervezési minta alkalmazásával érhető el az optimális programszerkezet kialakítása, ami biztosítani tudja a döntésredundanciák elkerülését.

A döntésarchiválást biztosító és a döntésszerkezetek optimális struktúra kialakításának eseteit taglaló tervezési minták során észre kell vennünk, hogy a döntések nem kiemelt állapota esetén is észlelhetők azok a döntéskapcsolatok, amelyek az adott tervezési mintára jellemzőek, így a tervezési minták nem csak a jelenlegi formájukban ismerhetők fel, hanem a döntések nem kiemelt állapota esetén a tervezési minták „lenyomatai" alapján is. **Tehát lehetőség van arra, hogy úgynevezett rejtett tervezési mintákat is felderítsünk.** A döntéskiemelés végrehajtásával az optimális struktúra kialakítható, ahol természetesen a döntésszerkezet ugyanaz marad, csak a döntések deklarálása már strukturáltabb formában történik.

Fontos tehát megjegyezni, hogy a tervezési minták jellemzése szerkezet függetlenül csupán a döntésszerkezet alapján is megtehető. Attól függően, hogy a döntések már osztályszerkezetbe kiemelten vannak deklarálva vagy még beágyazva, nem kiemelten deklaráltak másként jelennek meg a programban.

Ezzel új lehetőséget kapunk annak tisztázására, hogy milyen esetekben lehetséges egy adott tervezési minta felhasználása, hiszen egy tervezési minta leírásakor nem egy optimális megoldás paramétereit kell, hogy megadjuk, hanem egy olyan tervezési helyzet körülményeit, amellyel egy adott tervezési minta felhasználásával egy optimálisabb struktúra kialakítása érhető el. Tehát a tervezési minták bevezetésének lehetősége sokkal érthetőbbé válik, mivel konkrét megoldástól független jellemzők, a döntésszerkezetek szerint határozhatók meg a tervezési minták alkalmazhatósága.

Abban az esetben, ha a döntések elemzése automatizált eszközökkel is megtehető, akkor lehetőség nyílhat a tervezési minta alkalmazás szükségességének automatikus felismerésére.

Így a jelenlegi tervezési minta felismeréses megoldásokkal ellentétben nem csupán azt lehet észrevenni, hogy a kódban hol alkalmaztak valamilyen tervezési mintát – aminek gyakorlati haszna nem jelentős –, hanem azt, hogy hol lenne szükség tervezési minta alkalmazásával a meglévő függőségek, redundanciák feloldására.

Ezeknek megfelelően új eredményeknek a következőket tartom:

Tervezési minták jellemzése döntésszerkezetekkel

A tervezési minták jellemzését a tervezési helyzeteket leginkább kifejező döntés szerkezetekre alapozom. A tervezési minták felhasználhatósága így könnyebbé válik, mivel jellemzésük tervezési megoldás független módon, de kellő egzaktsággal tehető meg.

A „5.1 Tervezési minták döntés szerkezetek szerinti értelmezése" alfejezetben a tervezési minták döntésszerkezeteit mutatom be.

Az elképzelés alkalmazhatóságának igazolása céljából az „eFilter” kutatásfejlesztési projekt tervezési szakaszában a döntési struktúra elemzésével sikerült felfedezni néhány tervezési minta alkalmazhatóságát, melyek közül egy esetet „5.2. Sablonfüggvény tervezési minta döntésszerkezet szerinti felismerése az eFilter projekten” fejezetben, mint esettanulmányt ismertetek.

Tervezési minta felismerése, automatikus felismerés lehetősége döntésszerkezet alapján, megvalósítás függetlenül

A tervezési minták döntésszerkezete alapján lehetőség van arra, hogy a tervezési minták alkalmazhatóságának lehetőségét a programtervezők könnyebben észrevegyék.

A döntésszerkezet szerinti tervezési minta felismerés esettanulmánya a „5.3 Tervezési minta döntéskiemelés előtt és után” fejezetben található, ahol a „*Híd*” tervezési minta döntésszerkezetének megjelenését ismertetem döntés kiemelés és összevonás előtt és után. Egy példaprogrammal teszem érthetőbbé a „*Híd*” tervezési minta döntésszerkezeteinek megvalósulásait.

A tervezési minták döntésszerkezeteire alapozva a tervezési minták automatikus felismerése is támogatható úgy, hogy a megvalósítástól független döntésszerkezetek szerint tudjuk észrevenni a tervezési minták „lenyomatait” a tervben vagy forráskódban. Ezt a tervezési minta automatikus felismerés egy újszerű megközelítése lehet, amit új kutatási lehetőségnek tartok.

Tervezési minták osztályozása döntésszerkezetek szerint

A tervezési minták osztályozhatók aszerint, hogy milyen döntésszerkezet esetén adják meg az optimális döntésdeklarációkat, melyekkel a döntésredundancia elkerülése biztosítható.

A „5.1.4 Tervezési minták osztályozásának új módszere” fejezetben bemutatott osztályozást olyan új eredménynek tartom, ami hozzájárulhat ahhoz, hogy a tervezési minták döntésszerkezetből fakadó esetleges hasonlóságait észrevehessük, felhasználhatóságukat javítsuk.

Tervezési minták döntésszerkezeteinek JML alapú specifikációja

A JML lehetőséget ad arra, hogy a tervezési minták döntésszerkezeteit egzakt módon, a döntések reprezentációjának módjától függetlenül deklaráljuk. Ezzel megoldható a döntések viselkedési szerződéseinek összehasonlításával azon döntésszerkezetek felismerése, melyek az egyes mintákra jellemzők.

A tervezési minták JML alapú formalizálását a „*Híd*” tervezési minta döntésszerkezetének JML alapú specifikációjával szemléltetem: „5.4.1 Híd tervezési minta JML alapú formális leírása”, amit egy esettanulmánnyal teszek érthetőbbé a „5.4.2 Híd tervezési minta JML alapú formális leírása példán keresztül” fejezetben. Ezekre alapozva az „5.4.3 Tervezési minták döntésszerkezetek szerinti JML alapú formalizálásának szabályai” fejezetben összegzem tapasztalataimat.

A fejezetben először a tervezési minták osztályozása szerint ismertetjük a tervezési minták általános célkitűzéseit, majd összehasonlítjuk azokat az új elképzelés

szerinti célkitűzésekkel úgy, hogy a mintákat azok döntésszerkezete szerint jellemezzük. A fejezetben a tervezési minták szerkezetének ábrázolására használt OMT diagramokat [15] a kifejezőbb és jelenleg elterjedtebb UML diagramokkal helyettesítem. Mivel a tárgy szempontjából az osztályszerkezetek számítanak, ezért az egyes minták felépítését csak osztálydiagramokkal szemléltetjük. A döntésszerkezetek leírásához az azokat ábrázolni képes, viselkedést leíró aktivitás diagramokat [40] használom. Fontos megjegyezni, hogy az egyes döntésszerkezetek aktivitás diagramjain nem ábrázolom a teljes folyamatot, csak a döntésszerkezet szempontjából lényeges részeket, így az indító és befejező állapotok megjelenítésétől is esetenként eltekintek. A fejezetben csak azon tervezési mintákkal foglalkozok, melyek rendelkeznek értelmezhető döntésszerkezettel. (Az *„Egyke”, „Homlokzat”, „Emlékeztető”* tervezési minták leírásától eltekintek.)
Ezt követően a tervezési minták csoportosításának egy új módját ismertetem, ahol a döntések egymásba épülése szerint, illetve a döntés archiválás módozatai szerint határozhatunk meg csoportokat. A jelenleg meglévő osztályozás mellett az ismertetett csoportosítás egy párhuzamos megoldást jelent, ami hozzásegíthet a tervezési minták megértéséhez.
Lényeges résznek tekinthető az osztályozást követő tervezési minták döntéskiemelés előtti és utáni állapotával kapcsolatos jellemzést taglaló fejezet, ami után egy példán keresztül mutatom be a tervezési minta különböző állapotainak megjelenéseit.
Az előző fejezetben ismertetett JML-lel a tervezési minták szabályainak döntésalapú formális megfogalmazására teszek kísérletet, amit kiterjesztek a tervezési minták döntéskiemelés előtti és utáni állapotának JML alapú vizsgálatával. A tervezési minta döntéskiemelés előtti és utáni állapotának JML-es leírását egy példán keresztül szemléltetem.

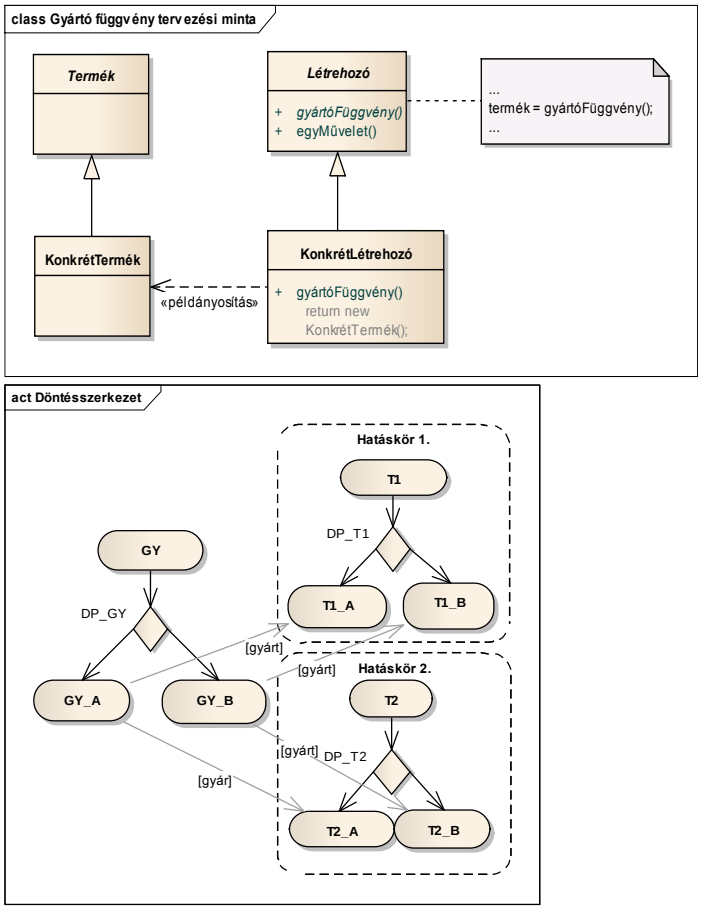
5.1 Tervezési minták döntésszerkezetek szerinti értelmezése

5.1.1 Létrehozási minták

Abban az esetben, ha egy döntés eredményét hatókörön kívül szeretnénk felhasználni, vagy a döntés eredménye nem használható fel az adott hatókörön belül a döntés egy másik döntéseseténél, illetve, ha az egyes döntési pontokon az adott döntésnek megfelelő, de új adattartalmat és metodológiát kell gyártani, akkor a döntésarchiváló tervezési minták adnak recepteket a probléma kezelésére.
Meg kell különböztetnünk gyártó és termék oldalt. Gyártó oldalon gyártó döntésekről, termék oldalon termék döntésekről van szó, amelyek természetesen a gyártó döntésektől függenek.

5.1.1.1 Gyártó függvény tervezési minta

Egy objektum létrehozásához biztosít absztrakt felületet, ahol a „gyártó” alosztályok határozzák meg, hogy melyik osztályt példányosítjuk.



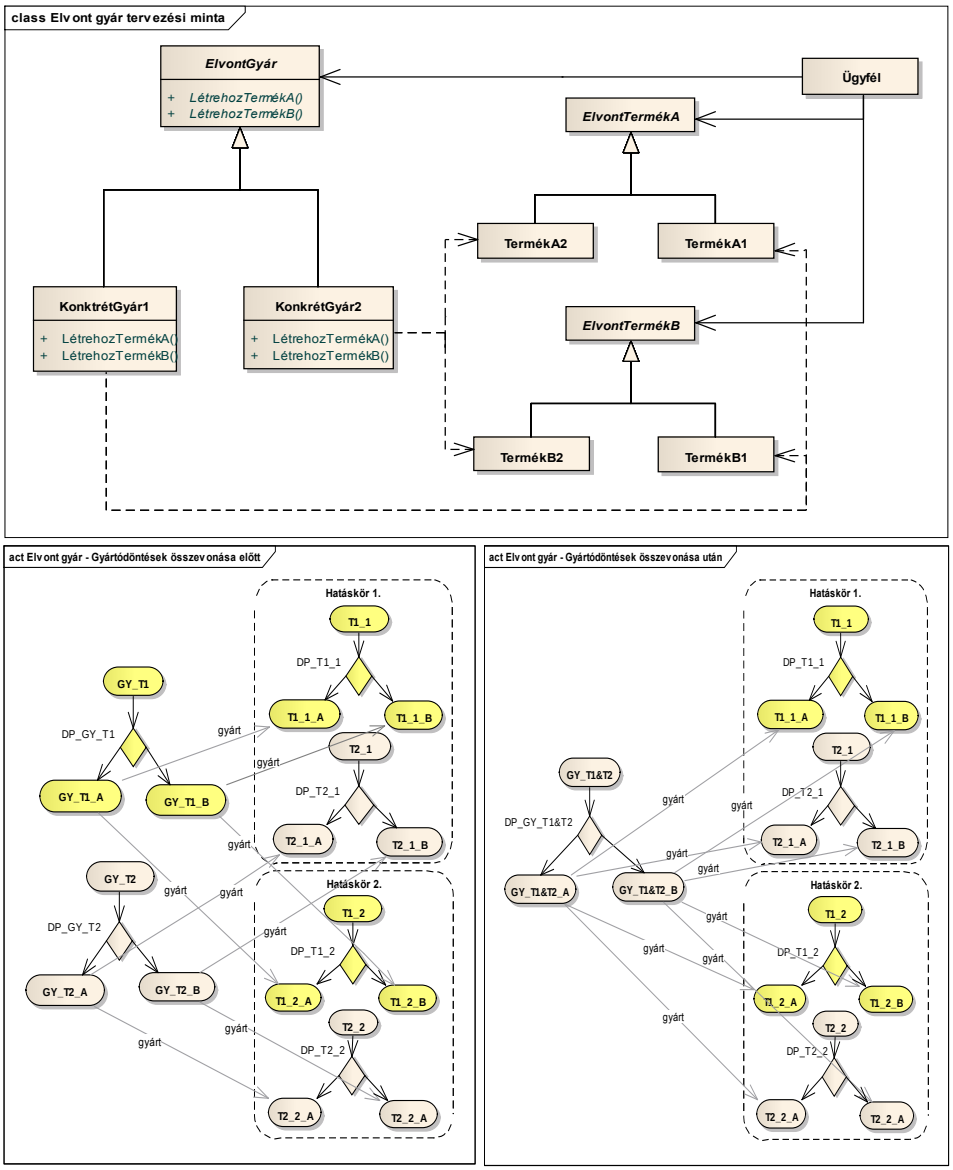
12. Diagram: *„Gyártó függvény”* tervezési minta osztálydiagram, döntésszerkezetek aktivitás diagramokkal

Döntésszerkezet szerinti jellemzés:
Abban az esetben használható, ha a döntéshozatal és a döntés felhasználása nem egy hatáskörben van, esetleg a döntést több hatáskörben is fel kell használni („*T1”, „T2”*), azaz a program futása során többször szükséges a gyártó döntés szerinti termékdöntés valamely döntési lehetőségét elérni.
Ha találunk a döntésösszevonás szabályai szerint összevonható, ekvivalens döntéspredikátumú és viselkedésű döntéseket („*T1”, „T2”*), amelyek összevonásuk után egy polimorf referenciával nem archiválhatók (pl. nem egy hatáskörben vannak vagy a döntéshozatal és a döntés felhasználása nem egy hatáskörben van), akkor feléjük egy gyártó döntést kell építeni, amely döntés döntéspredikátuma ekvivalens lesz az összevonandó döntések döntéspredikátumaival, azaz *DP_GY=DP_T1=DP_T2*.
Kiemelt állapotban ez azt jelenti, hogy a megfelelő *„KonkrétLétrehozó”* alosztályt (döntéslehetőséget) kiválasztva azt példányosítani kell, amit a *„Létrehozó”* szülőosztályra mutató referenciával (döntésarchiválás) lehet elérni.

A *„Gyártó függvény”* tervezési mintát felhasználva a gyártó oldali döntést csak egyszer kell meghozni azzal kapcsolatosan, hogy milyen terméket szeretnénk létrehozni, amit többször fel tudunk használni a szükséges termék döntések meghozására és azok bezárására.

5.1.1.2 Elvontgyár tervezési minta

Egymással kapcsolatban lévő, egymástól függő objektum családok létrehozására elvont felületet biztosít, úgy, hogy nem kell megadni, hogy az egymással függőségben lévő objektum családok mely típusa szerinti egyedeket akarunk létrehozni.

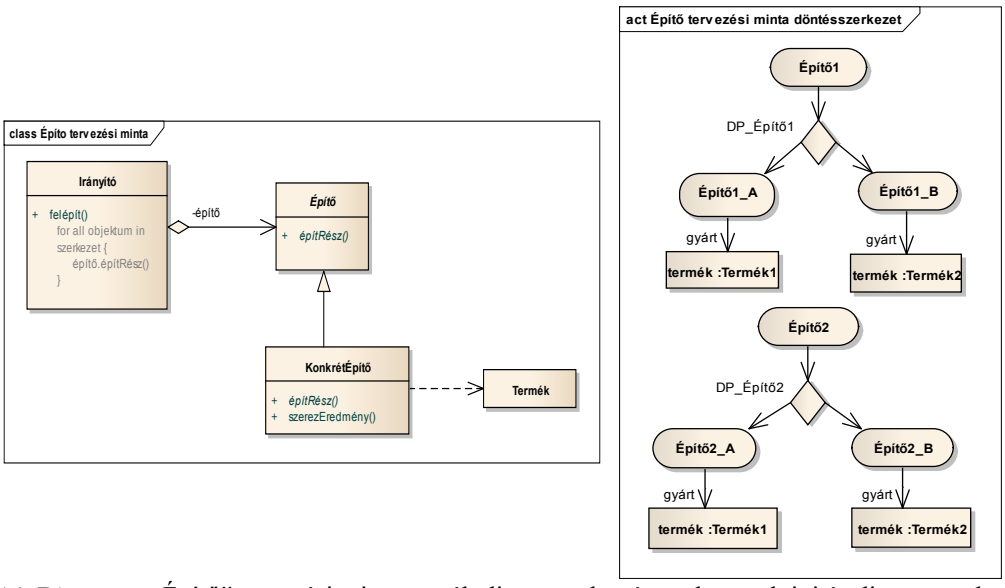


13. Diagram: *„Elvont gyár”* tervezési minta osztálydiagram, döntésszerkezetek aktivitás diagramokkal

Döntésszerkezet szerinti jellemzés:
Az „*Elvontgyár*” tervezési mintára úgy kell tekintenünk, mintha két „*Gyártó függvény*” tervezési minta gyártó oldalát olvasztanánk össze. Ez akkor lehet indokolt, ha valamilyen programszervezési ok miatt az ekvivalens döntéspredikátumú, de viselkedésileg nem ekvivalens termék oldali döntések nem vonhatók össze.
Ha vannak ekvivalens döntéspredikátumú gyártó döntéseink: *DP_GY_T1 =DP_GY_T2* (*„Gyártó függvény”* tervezési minta szerint két gyártó döntés), akkor azok összevonhatók egy gyártó döntéssé, ami után az összevont gyártó döntés meghozatalával mindkét termék döntés meghozható, amellyel lehetőség van arra, hogy a gyártó döntés szerint több termékdöntést „kezeljünk”.
Természetesen ilyenkor a termék oldali döntéseink döntéspredikátumai is ekvivalensek: *DP_T1_1=DP_T1_2=DP_T2_1=DP_T2_2*, amelyek egymással is ekvivalensek. Ugyanakkor fontos megjegyezni, hogy *„T1_1” – „T1_2”* és *„T2_1” – „T2_2”* döntéspárok viselkedésileg egymással nem ekvivalensek. Ha azok lennének, akkor egyszerűen egy *„Gyártó függvény”* tervezési minta használatával megoldható lenne a legyártásuk.

5.1.1.3 Építő tervezési minta

Az „*Építő*” tervezési minta célja az összetett objektum szerkezetek létrehozásakor az építési folyamat függetlenítése az objektumszerkezet ábrázolásától. A tervezési minta komplex objektum összetételek esetén nagyon jól használható arra, hogy összetett objektumszerkezetek gyártási folyamatát cserélhetővé tegyük.

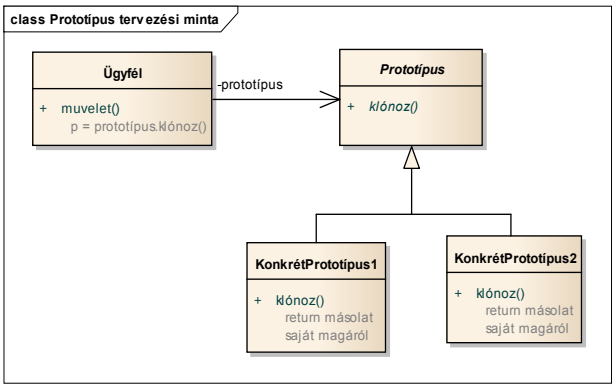


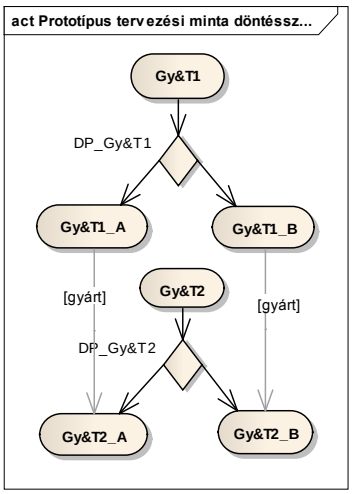
14. Diagram: *„Építő”* tervezési minta osztálydiagram, döntésszerkezet aktivitás diagrammal

Döntésszerkezet szerinti jellemzés:
Gyártó folyamat részeként a gyártandó termék típusára vonatkozó döntéseket tartalmaz, ahol a gyártandó termékek nem sorolhatók egy döntésstruktúrába.
A termék típusára vonatkozó döntések („*Építő1*”, „*Építő2*”) viselkedésileg ekvivalensek, azonban döntéspredikátumuk („*DP_Építő1*”, „*DP_Építő2*”) szerint nincs ilyen megkötés. Ennek megfelelően az egyező viselkedésű döntések összevonásával jöhet létre a tervezési mintára jellemző osztályszerkezet.
A különböző termék objektum szerkezetnek megfelelően más-más építési részfolyamatot kell használni, amelyre a döntés vonatkozik. Az építési folyamat váz szerkezete egységes a termék objektumszerkezet függő részeken kívül.

5.1.1.4 Prototípus tervezési minta

A „*Prototípus*” tervezési minta célja a „*Prototípus*” példány használatával meghatározni, hogy milyen típusú objektumot kell létrehozni az adott objektum lemásolásával.





15. Diagram: „*Prototípus*” tervezési minta osztálydiagram, döntésszerkezet aktivitás diagram

Döntésszerkezet szerinti jellemzés:
Ha a „*Gyártó függvény*” és „*Elvont gyár*” tervezési mintával ellentétesen a döntéshozatalt (gyártó döntés) és a döntés felhasználást (termék döntés) nem szükséges különválasztani, de a döntést több hatáskörben is fel kell használni (ld. 12. Diagram: „*T1*”, „*T2*”) úgy, hogy az eredeti „*T1*” döntéssel döntéspredikátumilag és viselkedésileg egyező, de új „*T2*” termék döntésre van szükség, akkor a „Gyártó függvény” tervezési mintától eltérően a gyártó döntést és termék döntést össze kell olvasztani. („*Gy&T1*”, „*Gy&T2*”)
Ha találunk a döntésösszevonás szabályai szerint összevonható, ekvivalens döntéspredikátumú döntéseket („*T1*”, „*T2*”), amelyek összevonásuk után egy polimorf referenciával nem archiválhatók (pl. nem egy hatáskörben vannak vagy a döntéshozatal és a döntés felhasználása nem egy hatáskörben van), akkor az egyiket

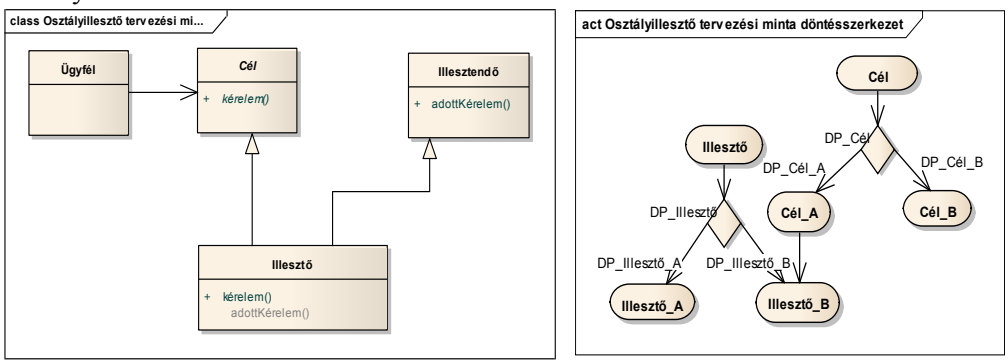
(„*T1*”) gyártó szerepkörrel felruházva felhasználhatjuk a vele viselkedésileg és döntéspredikátumában ekvivalens „*T2*” döntés legyártására.
Összefoglalva egy olyan speciális döntés archiváló tervezési minta, ahol a gyártó és a termék döntéshierarchiát összevonták. Mivel a gyártó és termék döntés döntéspredikátuma ekvivalens, így a döntésösszevonás szabályai szerint az összevonásuk lehetséges. Egy döntés szerint legyártott termék tud az adott döntésnek megfelelő további terméket gyártani. A „*Gyártó függvény*” tervezési mintához képest itt nem szükséges a gyártó és termék döntés konzisztenciáját megőrzendő azokat párhuzamosan karbantartani, mivel a két szerepkör „együtt él”.

5.1.2 Szerkezeti minták

A tervezési minták meglévő osztályozása szerint szerkezeti mintáknak elnevezett minta csoportot a döntések egymásba épülése, a döntések egymáshoz való viszonyulása szerint csoportosíthatjuk, osztályozhatjuk. A tervezési minták azonosíthatósága, csoportosíthatósága szempontjából lényegtelen, hogy az adott döntések melyik állapotukban definiáltak, azaz ki vannak-e emelve osztály hierarchiába vagy még metódusban vannak ágyazva. (Lásd: „5.3 Tervezési minták jellemzése döntéskiemelés előtt és után”)
Ez azt jelenti, hogy a tervezési minták detektálása döntéskapcsolatok, döntés szerkezetek alapján kell, hogy történjen. Tehát nem kiemelt döntések esetén is észrevehetők azok a döntéskapcsolatok, amelyek az egyes tervezési mintákra jellemzőek, melyek meghatározzák, hogy a döntésredundancia megszüntetése melyik tervezési minta szerinti struktúra felvételével valósulhat meg.

5.1.2.1 Illesztő tervezési minta

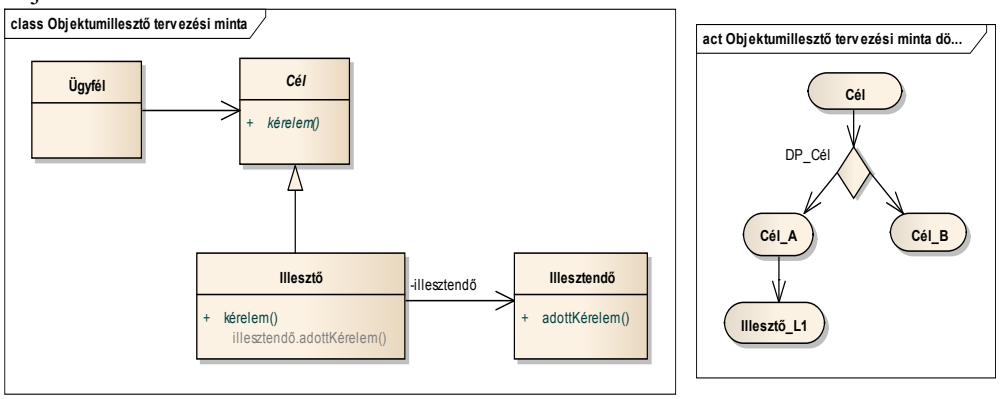
Az „*Illesztő*” tervezési minta célja, hogy egy osztály felületét az igények szerinti felületté alakítsa öröklődéses kapcsolattal (osztályillesztő esete) vagy objektum összetétellel (objektumillesztő esete). A „2.2 Objektumorientált tervezés alapelvei” fejezetben vázolt „Objektumösszetétel használata öröklés helyett, ahol lehet” tervezési elv tisztázza, hogy mely esetekben szükséges öröklődéssel vagy objektum összetétellel egy osztály által nyújtott szolgáltatást másik osztály számára elérhetővé tenni.
Osztályillesztő esete:



16. Diagram: „*Illesztő*” tervezési minta („*Osztályillesztő*”) osztálydiagram, döntésszerkezet aktivitás diagram

Döntésszerkezet szerinti jellemzés:
Két döntéshierarchia egy-egy döntéslehetőségének statikus összekapcsolását jelenti, amely akkor lehetséges, ha két döntés 1-1 döntéslehetőségének döntéspredikátuma ekvivalens.
Döntésszerkezet szerint *„Osztályillesztőről”* beszélünk, ha egy döntés („*Cél*”) döntéslehetősége („*Cél_A*”) tartalmaz olyan részt („*Illesztő_B*”), amely a döntésösszevonás szabályai szerint kiemelhető egy külön döntésszerkezetbe, illetve a tartalmazott rész „*Illesztő_B*” döntéslehetőségének döntéspredikátuma ekvivalens a tartalmazó döntéslehetőség („*Cél_A*”) döntéspredikátumával: *DP_Illesztő_B = DP_Cél_A*

Objektumillesztő esete:

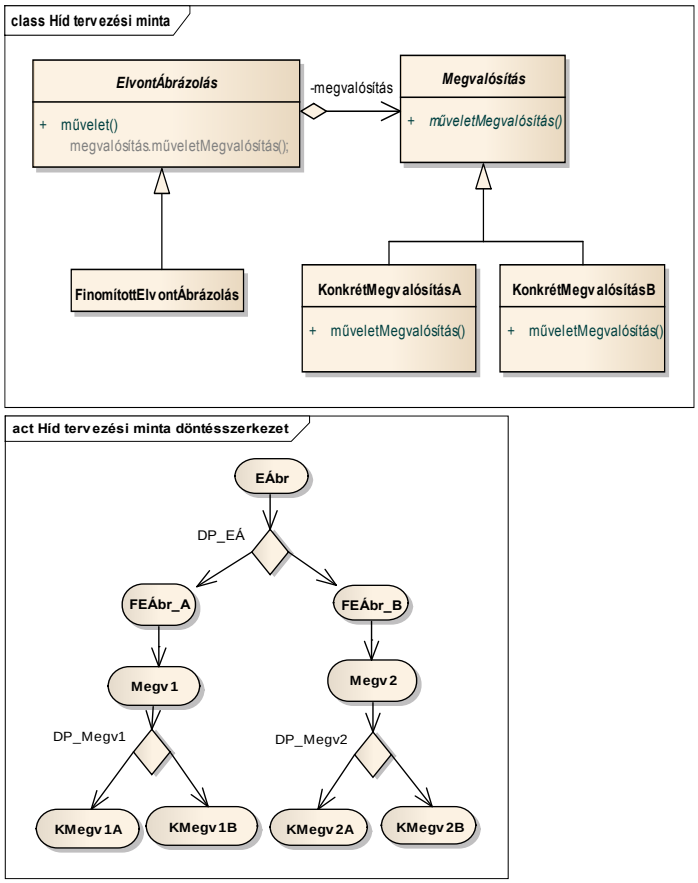


17. Diagram: *„Illesztő”* tervezési minta (*„Objektumillesztő”*) osztálydiagram, döntés-szerkezet aktivitás diagram

Döntésszerkezet szerinti jellemzés:
Döntésszerkezet szerint *„Objektumillesztőről”* beszélünk, ha egy döntés (*„Cél”*) valamely döntéslehetőségének (*„Cél_A”*) van olyan része (*„Illesztő_A”*), amit újra fel akarunk használni, de osztály struktúrába az *„Osztályillesztő”* tervezési minta követelményei szerint nem rendezhető.
Egy döntéshierarchia döntéslehetősége számára objektum összetétellel biztosítunk valamilyen szolgáltatást. Ez egy döntéslehetőség dinamikus kiegészítését jelenti.

5.1.2.2 Híd tervezési minta

A „*Híd*” tervezési minta célja a megvalósítás elvont ábrázolástól való különválasztása, hogy a kettő egymástól függetlenül kezelhető legyen.

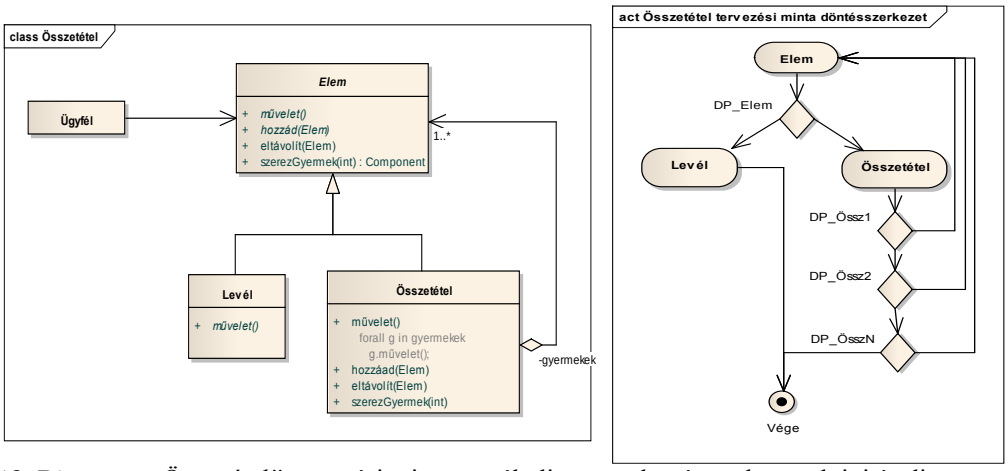


18. Diagram: *„Híd”* tervezési minta osztálydiagram, döntésszerkezet aktivitás diagram

Döntésszerkezet szerinti jellemzés:
A „*Híd*” tervezési minta esetén olyan összetett döntésszerkezetről van szó, ahol a tartalmazó („*EÁbr*”) döntéshez egy tartalmazott, szolgáltató döntést (*„Megv1”, „Megv2”*) rendelünk. A „döntés tartalmaz döntést” azon esete, ahol a tartalmazott döntések (*„Megv1”, „Megv2”*) összevonhatók, mert viselkedési szerződésük szerint ekvivalensek. (Megj.: Ha döntéspredikátum szerint lennének ekvivalensek, viselkedésileg nem, akkor a *„Látogató”* tervezési minta döntésszerkezetét kapnánk.)
A „*Híd*” tervezési minta esetén egy döntés nem döntés specifikus része tartalmaz egy másik döntést, amelynek viszont csak döntés specifikus része van.
Ebben az esetben kiemelt döntések esetén a tartalmazó döntést reprezentáló osztályhierarchia szülő osztályának lesz egy tag objektuma, amely objektumhivatkozásként a másik döntést reprezentáló osztályszerkezet absztrakt osztályára mutat. A „bezárt” tartalmazott döntés e felületen keresztül érhető el, azaz a tartalmazott döntés absztrakt osztálya adja a felületet, ami a döntésnek keretet ad. Felhasználása a felület által elérhető absztrakt metódus meghívásával lehetséges. A döntéslehetőségeket az alosztályok és az általuk implementált absztrakt metódus adja.

5.1.2.3 Összetétel tervezési minta

Az „*Összetétel*” tervezési minta olyan összetett objektum szerkezet létrehozására ad megoldást, amely fa struktúrába rendezi a komponenseket egységesen kezelve az elemi és összetett elemeket.

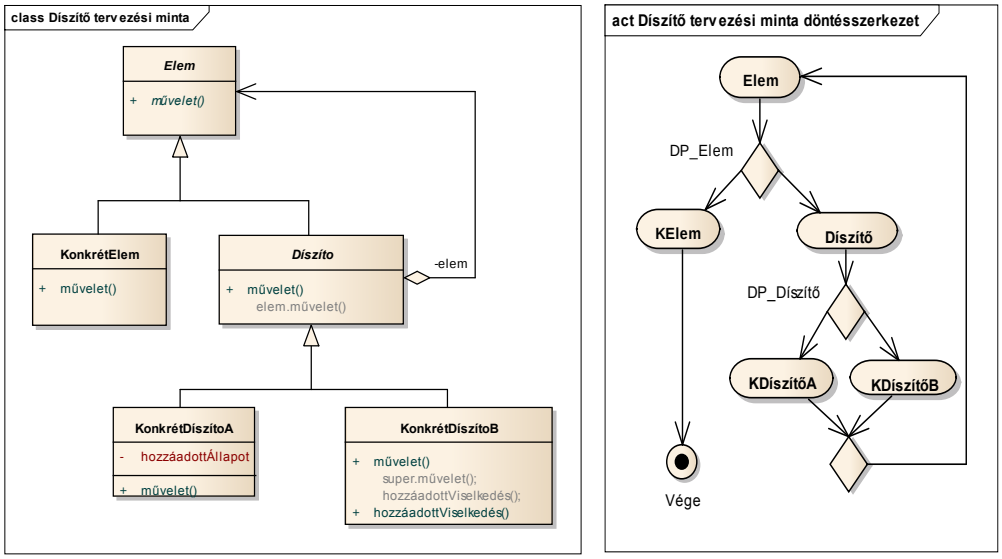


19. Diagram: *„Összetétel”* tervezési minta osztálydiagram, döntésszerkezet aktivitás diagram

Döntésszerkezet szerinti jellemzés:
Strukturált döntések speciális eseteként kezelhetők az úgynevezett rekurzív döntés struktúrák, ahol egy döntés (*„Elem”*) egyes döntéslehetőségeiben (*„Összetétel”*) mód van az eredeti döntés szerinti döntés meghozatalára úgy, hogy egy döntés döntéslehetőségébe több eredeti döntés viselkedésével megegyező döntés is beágyazható. A *„DP_Össz1”, „DP_Össz2”, „DP_Össz3”* döntéspredikátum szerinti döntések határozzák meg, hogy van-e az eredeti *„Elem”* döntés szerinti további döntés. (Lásd objektum-összetétel számossága!) Szükséges olyan döntéslehetőség is, amely „terminál”, azaz nem tartalmaz az eredeti döntésnek megfelelő további döntést. (*„Levél”*)

5.1.2.4 Díszítő tervezési minta

Objektumok új tulajdonságokkal, viselkedéssel való kiegészítésének módja, amellyel elkerülhető, hogy a különböző követelményekkel, viselkedéssel való kiegészítés esetén minden egyes osztályra alosztályként kelljen bevezetni az új követelményeket. Elegendő az új viselkedésekre létrehozott alosztályokat létrehozni, amelyek elvégezve a kiegészítő tevékenységet tovább delegálják a működést azon alosztályok objektumainak, amelyek kiegészítését el kellett végezni. Ezekkel az eredeti alosztályok szerinti példányok dinamikusan ruházhatók fel új viselkedéssel.



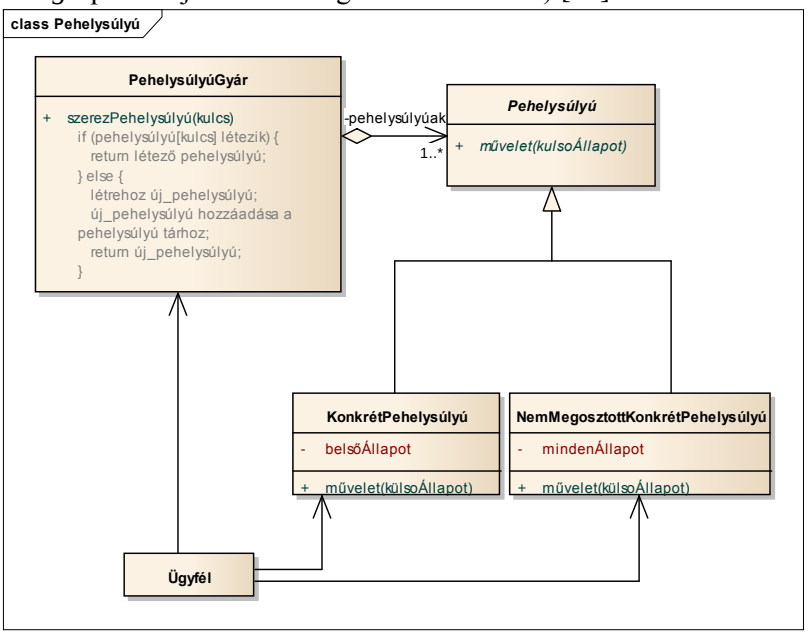
20. Diagram: *„Díszítő”* tervezési minta osztálydiagram, döntésszerkezet aktivitás diagram

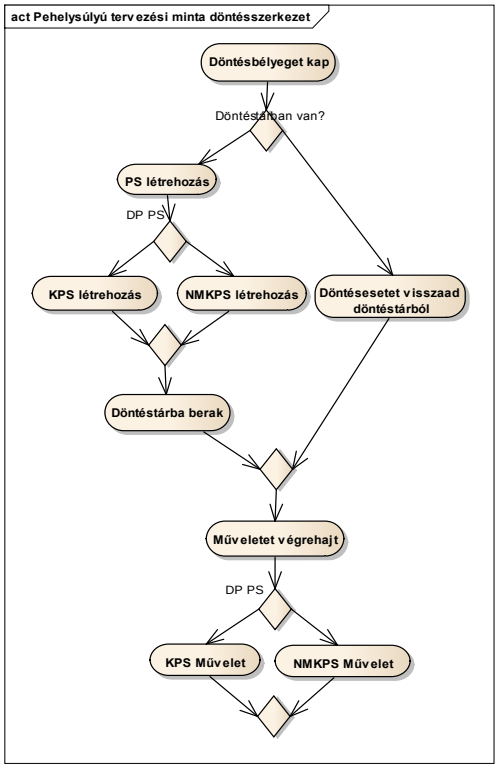
Döntésszerkezet szerinti jellemzés:
Lehetőséget ad egy döntéshierarchia (*„Elem”*) döntéslehetőségeinek dinamikus kiegészítésére úgy, hogy az egyik döntéslehetőségbe ágyazott döntés *(„Díszítő”*) szerinti viselkedéssel kiegészíthetők az eredeti döntéshierarchia döntéslehetőségei.
A strukturált döntések speciális eseteként úgynevezett rekurzív döntés struktúraként bevezetett „*Összetétel*” tervezési minta specializációjaként értelmezhető, amely a fentebbiekben leírtak szerint az „*Összetétel*” tervezési mintát egészíti ki azzal, hogy az egyik döntéslehetőséget nem csak arra „készíti fel”, hogy az eredeti döntéshierarchia szerinti döntést tartalmazzon, hanem az eredeti döntést egy másik döntéssel is kiegészíti. Fontos ugyanakkor megemlíteni, hogy az „*Összetétel*” tervezési mintával ellentétben a „*Díszítő*” tervezési mintában lévő tartalmazó döntés csak egy tartalmazott döntést tartalmazhat, azaz nem alkalmas „döntésfa” struktúra kialakítására, hanem csupán valamilyen lista adatszerkezetet, döntésláncot lehet vele ábrázolni. (Lásd objektum-összetétel számossága!)
A „*Díszítő*” a döntésösszevonás leírásakor részletezett azon esetet írja le, amikor egy összetett döntésszerkezet esetén a tartalmazó döntés döntéslehetőségei nem ekvivalensek a tartalmazott döntés döntéslehetőségeivel, hanem annak kiterjesztései. Az adatszerkezetet tekintve az adott döntés által érintett adatszerkezettel leírt állapot halmaznak a kiegészítő döntés adatszerkezete egy részhalmazát adja meg, illetve az adott döntés viselkedését megadó utófeltételeivel a kiegészítő döntés döntéslehetőségeinek utófeltételeiben vannak ekvivalens részek, amelyek további utófeltételekkel egészülnek ki.

5.1.2.5 Pehelysúlyú tervezési minta

A „*Pehelysúlyú*” tervezési minta a nagy finomságú objektumok hatékony felhasználását kezeli úgy, hogy hozzáférésükre egy közös kezelést vezet be.

Olyan hatékonyságnövelő megoldás, amit a jelenleg egyik leginkább elterjedt programozási nyelv a Java alapeszközei számos esetében használnak. (Például *String* típusú objektumok megosztott kezelése.) [46]



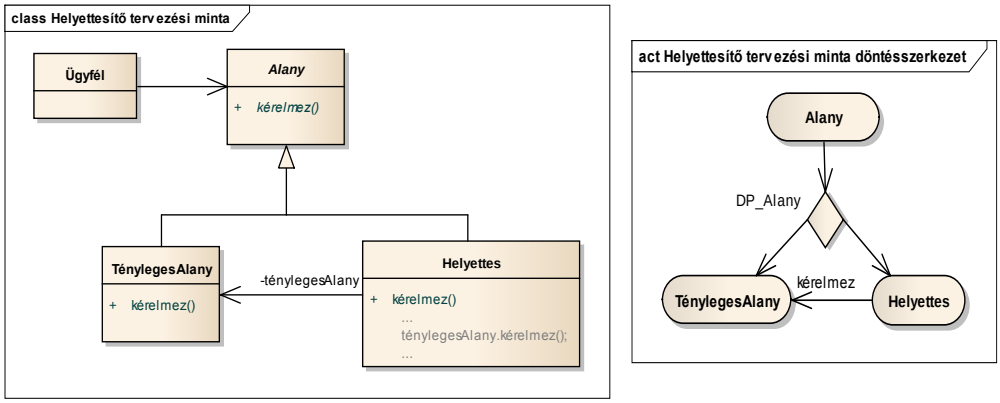


21. Diagram: *„Pehelysúlyú”* tervezési minta osztálydiagram, döntésszerkezet aktivitás diagram

Döntésszerkezet szerinti jellemzés:
A „*Pehelysúlyú*” tervezési minta az eredeti értelmezés szerint szerkezeti tervezési minták közé sorolható, de a döntés szerkezet és viselkedés szerinti az úgynevezett döntés archiváló minták közé tartozik, ahol jelenleg a „*Pehelysúlyú*” mintát leszámítva létrehozási tervezési minták vannak.
A „*Pehelysúlyú*” tervezési minta célja korábbi döntések eredményeinek felhasználása, azaz későbbi döntések kiváltása.
Az előzőekben bemutatott döntésarchiváló tervezési mintáktól annyiban különbözik, hogy bevezet egy döntéstárat, ahol az archivált döntéseket egy döntésbélyeggel azonosítja.
Amikor szükség van valamelyik döntéslehetőség szerinti döntésre, akkor megvizsgáljuk, hogy az adott döntésbélyeggel rendelkező döntéseset már létezik-e a döntéstárban (*„Döntéstárban van?”*). Ha igen, akkor újra felhasználható, vagyis a megfelelő döntéseset szerinti példányt kivesszük a „döntéstárból”.
Ha nem volt az adott döntéslehetőségnek és „döntésbélyegnek” megfelelő korábbi döntéseset, azaz nincs megfelelő példány a döntéstárban, akkor létre kell hozni egyet és az kerül az adott ponton felhasználásra, és bekerül a döntéstárba is. (*„PS létrehozás”*)
Az archiváló tervezési minták *„Gyártó függvény”*, *„Elvontgyár”* tervezési mintáihoz hasonlóan, ahol a gyártó döntés döntéspredikátuma és a termék döntés döntéspredikátuma ekvivalens, egy döntésbélyeg szerinti döntés létrehozásakor a döntéspredikátuma ekvivalens lesz az adott döntésbélyeg szerint összekötött későbbi felhasználási helyek, döntésesetek döntéspredikátumával. (A döntés döntéstárba archiválásának döntéspredikátuma – *„DP PS”* megegyezik a felhasználás döntéspredikátumával.) Az egyező döntésbélyeggel rendelkező döntések döntéspredikátuma ekvivalens.

5.1.2.6 Helyettes tervezési minta

A „*Helyettes*” tervezési minta célja, hogy lehetőség legyen egy objektumot egy képviselőn keresztül elérni. Akkor szükséges alkalmazni, ha az irányítani kívánt objektumot biztonsági megfontolásból nem akarjuk közvetlenül elérhetővé tenni, illetve, ha az adott objektum nem elérhető az adott környezetben, hanem felé egy speciális interfészen keresztül biztosítható a hozzáférés (távoli objektum elérés).
A „*Helyettes*” használata hatékonyságnövelés szempontjából is fontos, mivel lehetőséget ad a költséges objektumok esetén azok létrehozását a konkrét felhasználásig elhalasztani. Addig a helyettesén keresztül jelenik meg csak a rendszerben.



22. Diagram: *„Helyettesítő”* tervezési minta osztálydiagram, döntésszerkezet aktivitás diagram

Döntésszerkezet szerinti jellemzés:
A *„Helyettesítő”* tervezési minta döntésszerkezete szerint van egy döntéslehetőség, amely tovább delegálja a „kérést” egy másik döntéslehetőség felé. Ez egy döntés döntéslehetőségeinek statikus összekapcsolását jelenti, ahol az egyik döntéslehetőség viselkedésileg ekvivalens lesz a döntés egy másik döntéslehetőségének viselkedésével.
Az értelme az, hogy a „Helyettes” kevesebb erőforrást igényel (csak az interfészt biztosítja), mint a „*TénylegesAlany*” és így az archivált döntés végrehajtásáig elhalasztható a „költséges” „*TénylegesAlany*” döntéslehetőség végrehajtása.
Használata döntésszerkezetileg az előzőekben megadott módon értelmezhető, ugyanakkor alkalmazása a döntésösszevonás szabályai szerint nem vizsgálható.
Ugyanakkor érdekes megközelítés lehet a döntéslehetőségek erőforrásigény szerinti súlyozása, ami alapján, ha egy döntéslehetőség erőforrásköltsége magas, bevezetése indokolt lehet.

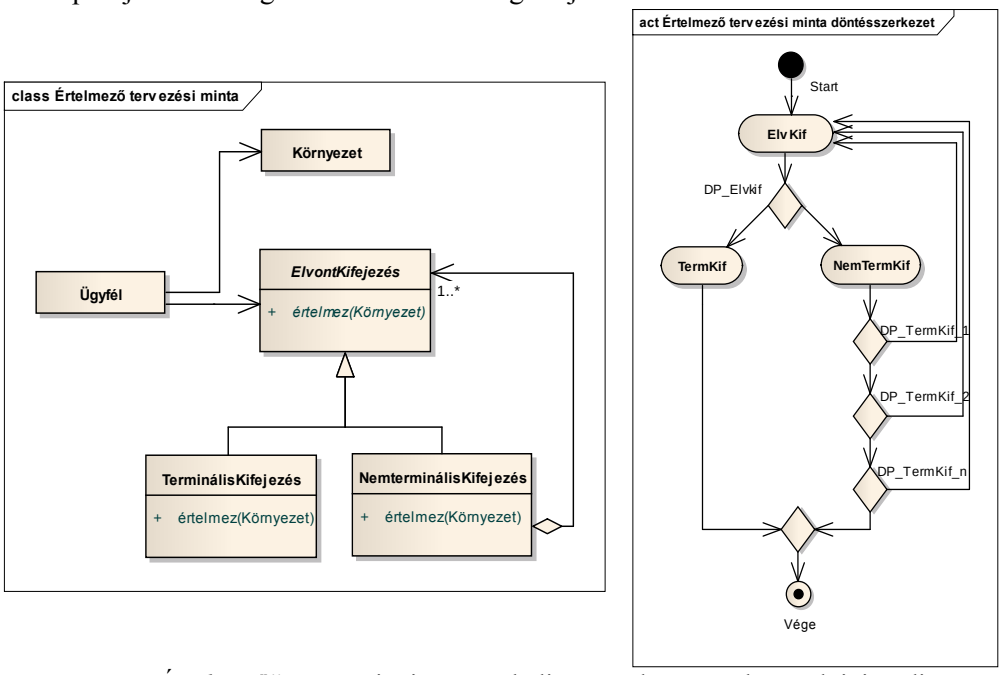
5.1.3 Viselkedési minták

A nevükből következően a rendszerek viselkedésének modellezésére adnak megoldásokat úgy, hogy nem csak az osztályok és objektumok szerkezetét specifikálják, hanem azok együttműködését és kommunikációját is.
A viselkedési mintákat a szerkezeti mintákhoz hasonlóan a döntés szerkezetek alapján vizsgálhatjuk és osztályozhatjuk.

5.1.3.1 Értelmező tervezési minta

Egy nyelv nyelvtanát ábrázolni képes megoldás, amellyel a nyelv mondatait a megadott reprezentációval ábrázolni és azon keresztül értelmezni tudjuk.
Az „*Értelmező*” tervezési minta osztályszerkezetén jól látszik, hogy a kifejezéseknek két fajtáját különböztetjük meg: összetett és elemi kifejezések. A nyelv mondatainak ábrázolásakor a létrejövő kifejezés fák levél elemei elemi kifejezések – terminális elemek lesznek, a fa többi csomópontja összetett kifejezés – nem terminális elem, amely egyes típusaira külön alosztályokat is bevezethetünk.

Megkötés, hogy lennie kell legalább egy elemi kifejezéseket ábrázolni képes alosztálynak, különben nem lennének terminális elemei az adott nyelv kifejezésfáinak, illetve legalább egy nem csomópont típusú alosztálynak, hiszen vele tudunk fát építeni.
Az „*Összetételhez*” hasonlóan fa struktúra felépítését teszi lehetővé, amely csomópontjaihoz a megfelelő viselkedést ragasztja.



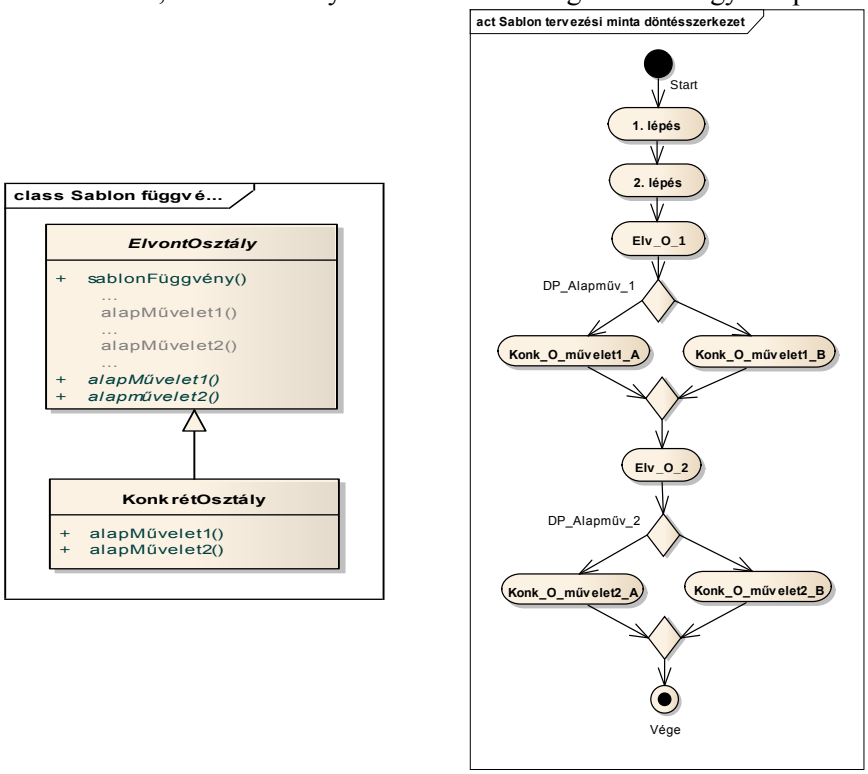
23. Diagram: *„Értelmező”* tervezési minta osztálydiagram, döntésszerkezet aktivitás diagram

Döntésszerkezet szerinti jellemzés:
„Viselkedési mintaként” azonosították, de a szerkezeti mintákra, azokon belül is döntésszerkezet szerint az *„Összetétel”* tervezési mintára hasonlít. Ennek megfelelően a döntésszerkezet szerinti osztályozás alapján az „*Értelmező*” és az „*Összetétel*” természetesen egy csoportba kerül.
Az *„Összetétel”* tervezési mintához hasonlóan a strukturált döntések speciális eseteként úgynevezett rekurzív döntés struktúrák létrejöttének lehetőségét adja, ahol a döntéshierarchiában lennie kell olyan döntéslehetőségnek (*„NemTermKif”*), amelyben lehetőség van több (lásd objektum-összetétel számossága!) az eredeti döntéshierarchiával egyező döntés meghozatalára *(„DP_TermKif1”*, *„DP_TermKif2”*, *„DP_TermKif3”*), illetve szükséges olyan döntéslehetőség is, amely „terminál”, azaz nem tartalmaz eredeti döntés szerinti struktúrát (*„TermKif”*).

5.1.3.2 Sablonfüggvény tervezési minta

A „*Sablonfüggvény*” tervezési minta esetén egy algoritmus vázát készítjük el egy absztrakt szülőosztályban, amely egyes lépéseit megvalósító metódusokat az alosztályba delegáljuk. Az alosztályokkal az egyes lépések megvalósításai

cserélhetők. A „*Sablonfüggvény*” tervezési minta az egyik objektumorientált alapfogalom, a „Inversion of Control”-ra, azaz „Vezérlés inverzió” teljesülésére ad receptet, amely szerint a szülőosztály határozza meg a vezérlést, vagyis azt, hogy mit mikor hívunk, és az alosztályokban kerülnek megadásra az egyes lépések.

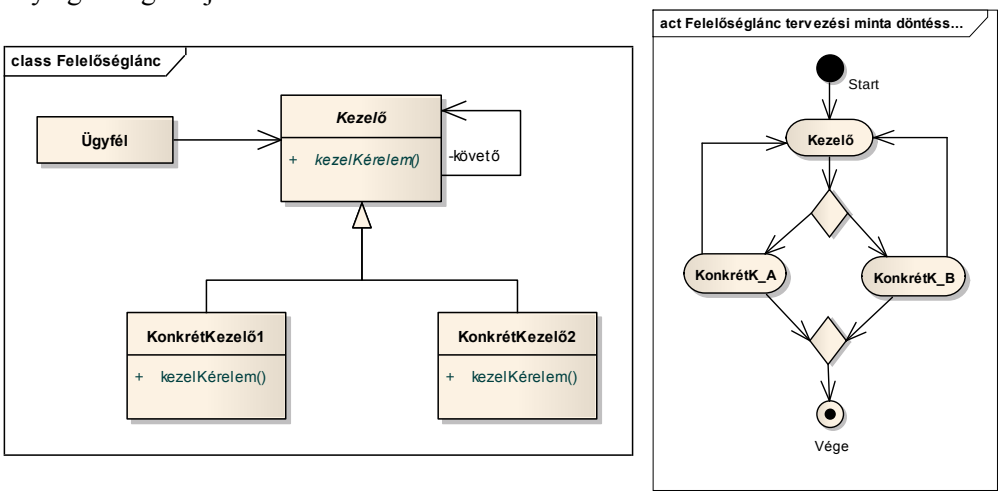


24. Diagram: *„Sablonfüggvény”* tervezési minta osztálydiagram, döntésszerkezet aktivitás diagram

Döntésszerkezet szerinti jellemzés:
A „*Sablonfüggvény*” tervezési minta döntésszerkezete esetén egy algoritmusban vannak döntésfüggő részek, azaz az egyes alapműveletekre vonatkozó döntésfüggő részek („*Elv_O_1*”, „*Elv_O_2*”) külön döntésekként értelmezhetők, amelyek döntéspredikátumai ekvivalensek.
Az ekvivalens döntéspredikátumú döntéseket (*DP_Alapműv_1=DP_Alapműv_2*), amelyek döntéslehetőségei az alapműveletek variációit deklarálják (*„Konk_O_művelet1_A”, „Konk_O_művelet1_B”, „Konk_O_művelet2_A”, „Konk_O_művelet2_B”*), egy osztályszerkezet alosztályaiba összevonva kell deklarálnunk („KonkrétOsztály”). („3.2 A döntésredundanciák elkerülésének esetei” fejezet ekvivalens döntéspredikátumú döntések összevonása eset.) Felhasználáskor a döntéspredikátum szerinti döntéslehetőség alosztályának példányával érjük el a megfelelő konkrét alapműveleteket, melyek a keretet adó „sablonfüggvényt” a döntésnek megfelelő tartalommal töltik fel.

5.1.3.3 Felelősséglánc tervezési minta

A „*Felelősséglánc*” tervezési minta célja, hogy egy kérelem küldőjének a tényleges végrehajtóhoz való kötését elkerüljük. Ennek kezelése úgy lehetséges, hogy egy felületen keresztül a kérelem kezelésére több típusú alosztálynak-implementációnak is jogot adnunk. Az egyes alosztályok különböző kezelési módszereket alkalmazhatnak, vagy bármelyikük tovább delegálhatja a kérelmet egy másik „*KonkrétKezelő”-nek* abban az esetben, ha a kezelést nem tudja megoldani. Dinamikusan kerül meghatározásra, hogy a kérelmet melyik alosztály szerinti objektum fogja tovább delegálni (nem terminális csomópont), és melyik fogja lekezelni (terminális csomópont).
Olyan lista felépítését idézi elő, ahol az egyes résztvevők a terminális elem, azaz a tényleges végrehajtó felé mutatnak.

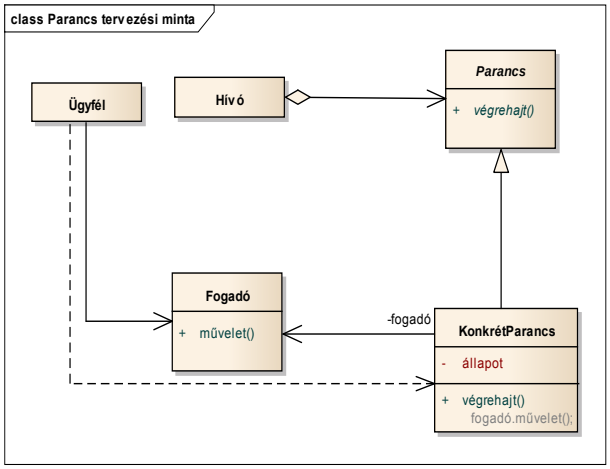


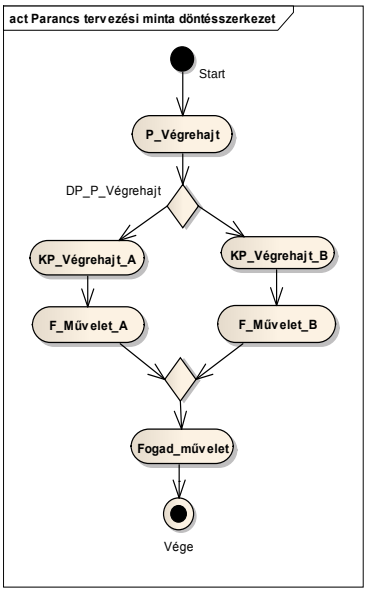
25. Diagram: *„Felelősséglánc”* tervezési minta osztálydiagram, döntésszerkezet aktivitás diagram

Döntésszerkezet szerinti jellemzés:
A „*Felelősséglánc*” tervezési minta döntésszerkezete szerint olyan döntést tartalmaz, amely egy kérelem különböző kezelésének módszereit taglalja. A rekurzív strukturált döntésszerkezetek egy esete, mivel bármelyik kérelem-kezelési döntéslehetőség (*„KonkrétK_A”, „KonkrétK_B”*) rekurzívan az adott döntésnek megfelelő további döntést tartalmazhat. Ezzel az „*Összetétel*” tervezési mintával ellentétben nem fa döntésszerkezet, hanem a „*Díszítő*” mintához hasonló döntéslánc keletkezhet, mivel egy döntés döntéslehetőségei csak egy-egy a tartalmazó döntéssel megegyező döntést tartalmazhatnak.
Tehát a különböző kérelemkezelések mellett dinamikusan kerül meghatározásra, hogy a kérelem továbbdelegálása megtörténik-e vagy sem, vagyis nem döntéslehetőség függő.
A „*Felelősséglánc*” tervezési minta tehát a rekurzív döntés struktúrák létrehozásának egy esete a „*Díszítő*”, „*Összetétel*” és „*Értelmező*” tervezési minták mellett.

5.1.3.4 Parancs tervezési minta

A „*Parancs*” tervezési minta célja, hogy a „parancsot” a hívó és a fogadó között cserélhetővé tegyük úgy, hogy a két fél ne is legyen kötve.
A „*Parancs*” kezelésére a minta egy olyan felületet biztosít, amely a fogadó felet képviseli (tárolja), illetve „elrejti” a tényleges művelet algoritmusát, annak elérésére a hívó fél felé csak egy absztrakt felületet biztosít. Arról, hogy milyen parancs végrehajtása várható, csak az „*Ügyfél*” tud.



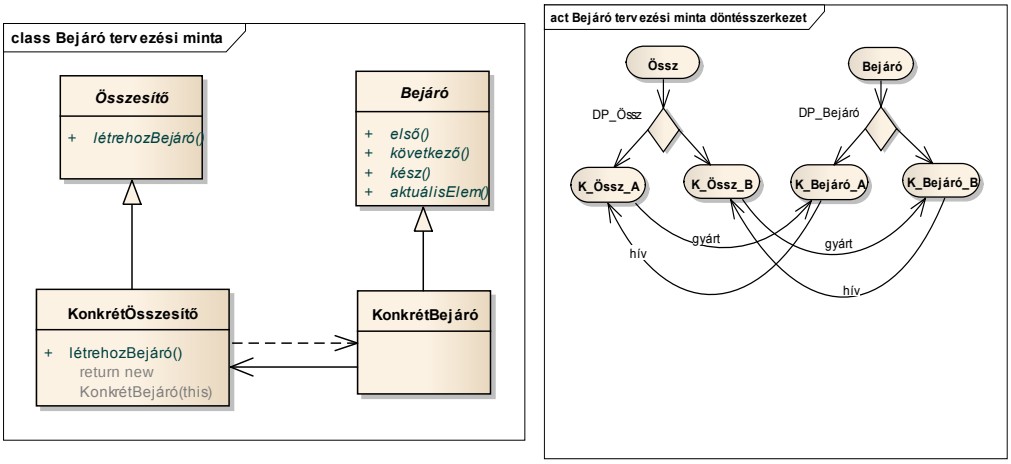


26. Diagram: *„Parancs”* tervezési minta osztálydiagram, döntésszerkezet aktivitás diagram

Döntésszerkezet szerinti jellemzés:
A „*Parancs*” tervezési minta döntésszerkezete szerint olyan döntést (*„P_Végrehajt”*) tartalmaz, amely a „*Hívó*” és egy „*Fogadó*” fél közötti különböző „parancsok” közül választ (*„KP_Végrehajt_A”*, *„KP_Végrehajt_B”*), tehát a „*Parancs*” döntéshierarchiája a hívó által a hívott oldalán végrehajtandó műveletek lehetőségeit taglalja. „Futás” közben dől el a döntéshierarchia döntésesetei során, hogy a megadott döntéslehetőségek közül melyik műveletet választják ki és hajtják végre, amely a hívó és a hívott számára rejtve marad.
Ha találunk a döntésösszevonás szabályai szerint összevonható olyan döntéseket (*„P_Végrehajt”* döntésesetei), amelyek egy *„Hívó”* részéről a *„Fogadó”*-n végrehajtott műveletekről döntenek, akkor a döntésösszevonás transzformációk elvégzése után a *„Hívó”* és *„Fogadó”* fél közötti műveleti lehetőségeket egy döntéshierarchia döntéslehetőségeiként (*„Parancs”*) egy osztályhierarchiába definiálhatjuk.

5.1.3.5 Bejáró tervezési minta

A „*Bejáró*” tervezési minta összetett objektumszerkezetek elemeinek soros eléréséhez biztosít absztrakt felületet úgy, hogy a bejárandó objektumszerkezetet is egy absztrakt felületen keresztül látjuk.



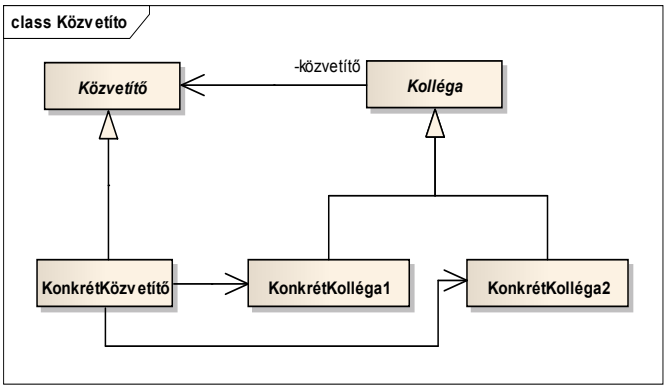
27. Diagram: Bejáró tervezési minta osztálydiagram, döntésszerkezet aktivitás diagram

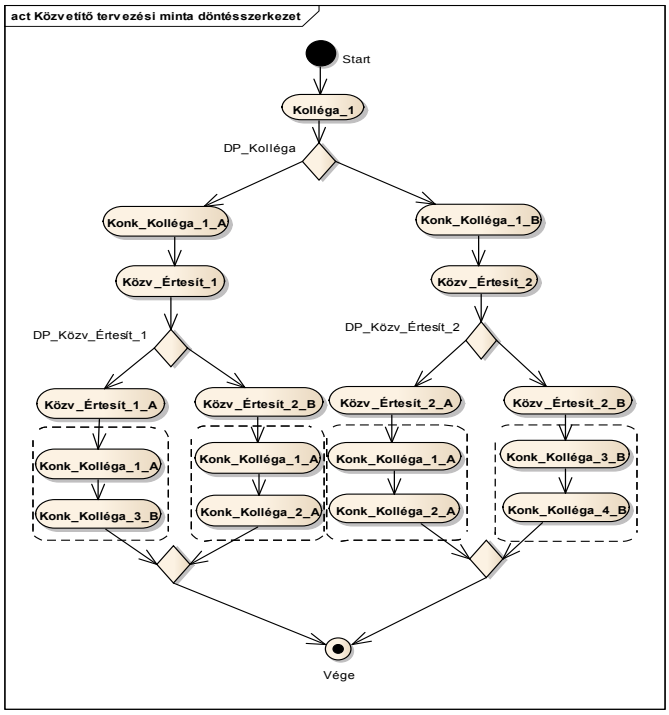
Döntésszerkezet szerinti jellemzés:
Összetett döntésstruktúraként kerül összekapcsolásra két döntéshierarchia (gyártó és egyben bejárandó objektumokat kezelő döntéshierarchia – *„Össz”*, bejáró döntéshierarchia – „*Bejáró*”) úgy, hogy a két döntéshierarchia döntéslehetőségei vannak összekötve. Az egyik irányból *„Össz”* döntés döntéslehetőségei legyártják a „*Bejáró*” döntés döntéslehetőségeit, a másik irányból „*Bejáró*” döntés döntéslehetőségei biztosítják az *„Össz”* döntéslehetőségei szerinti adatszerkezetek bejárását.
Ebben az esetben tehát egy döntés („*Bejáró”*) tartalmaz egy olyan másik döntést („*Összesítő*”), amely döntéspredikátuma ekvivalens a saját döntéspredikátumával.
Az „*Összesítő*”-*„Össz”* döntéshierarchia döntéslehetőségei (*„K_Össz_A”*, *„K_Össz_B”*) a „*Gyártó függvény*” tervezési mintához hasonlóan legyártják a „*Bejáró*” döntéshierarchia nekik megfelelő döntéslehetőségek szerinti „*Bejárókat*” (*„K_Bejáró_A”*, *„K_Bejáró_B”*). Ennek megfelelően a két döntés döntéspredikátuma ekvivalens lesz (*DP_Össz = DP_Bejáró*). Az „*Összesítő*” döntés döntéslehetőségei meghatározzák azt, hogy milyen „*Bejáró*” döntéslehetőséggel lehet az adott „*KonkrétÖsszesítő*” szerkezet elemeit bejárni.
A „*Bejáró*” döntés az adott „*Összesítő*” döntés szerinti objektum szerkezet elemeinek bejárására ad megoldásokat, azaz döntéslehetőségei a különböző objektumszerkezetek bejárási megoldásait fogalmazzák meg.
Kiemelt állapotban felhasználáskor, azaz az egyes döntéseseteknél mind a „*Bejáró*” mind a „*Összesítő*” döntés archiválva lesz, azaz nem ismert, hogy melyik „*Összesítő*” döntéslehetőség szerinti objektumszerkezet melyik „*Bejáró*” döntéslehetősége szerint kerül bejárásra.

5.1.3.6 Közvetítő tervezési minta

A „*Közvetítő*” tervezési minta célja, hogy objektumok egy halmazának együttműködését egy objektumon keresztül irányítsa.
Objektum halmazok résztvevői közötti „laza csatolás” megvalósításának az eszköze, mellyel az egyes objektumokat nem kell összekapcsolni, hanem azok egy

„kiemelt” szerepkörű „közvetítő” objektumon keresztül vannak egymással kapcsolatban, értesülnek egymás állapotváltozásairól.
A tervezési minta két elvont felületet tartalmaz. Az egyik felület a „*Kolléga”* objektumokat képviseli, a másik a „*Közvetítő”* objektumokat.
Az együttműködő objektumokat egy közös felület alá kell rendelni („*Kolléga”*). A különböző „*Kolléga”* típusú objektumok halmazára külön-külön „*KonkrétKözvetítő”*-ket vezethetünk be, amelyek az egyes konkrét „*Kolléga”* objektum halmazok együttműködését biztosítják.
Tehát a „*KonkrétKözvetítők”* a kapcsolódó „*Kolléga”* objektumok közötti egyes függőségi viszonyok konkrét megvalósulásai.



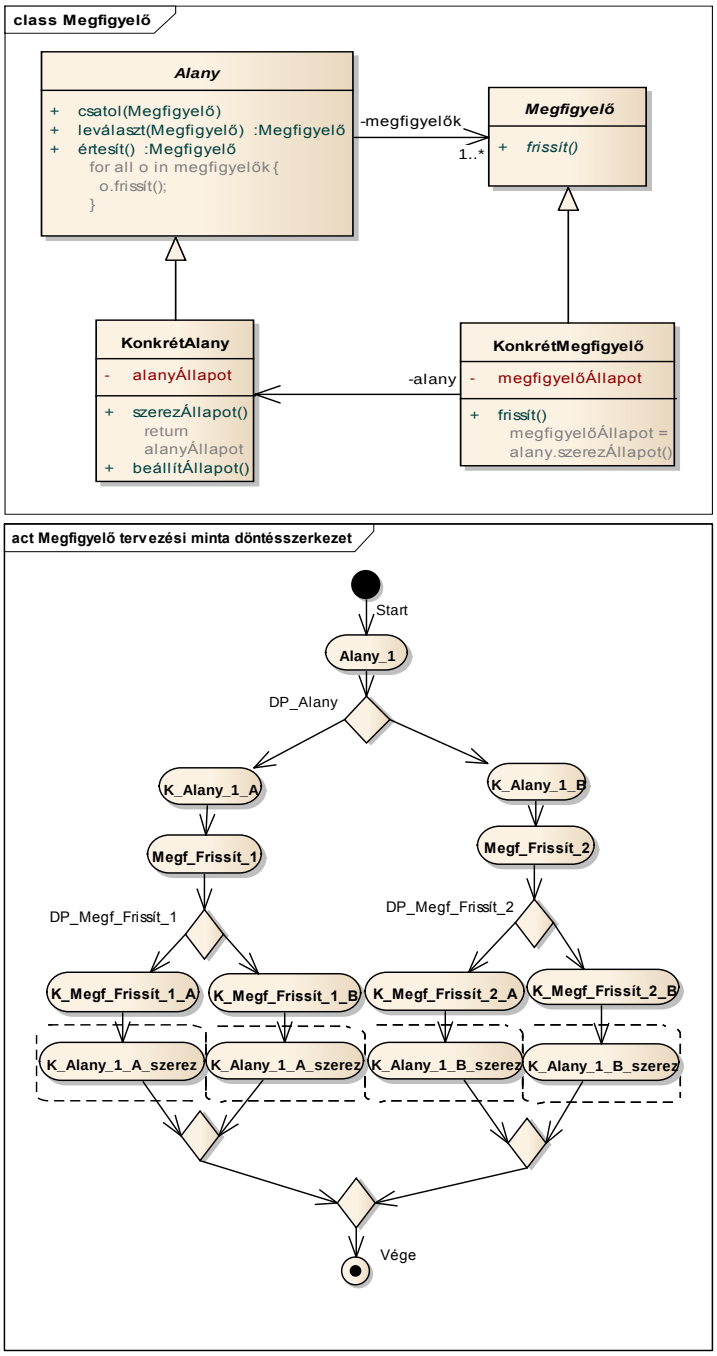


28. Diagram: Közvetítő tervezési minta osztálydiagram, döntésszerkezet aktivitás diagram

Döntésszerkezet szerinti jellemzés:
Összetett döntésstruktúra, ahol két döntéshierarchia több döntésesetének együttműködését kell megszervezni: *„Közvetítő”* - *„Közv_Értesít_1”, „Közv_Értesít_2”* döntéshierarchia, *„Kolléga” – „Kolléga_1”* döntéshierarchia, amely beégetett döntéslehetőségei, mint összehangolni kívánt döntésesetek jelennek meg: *„Konk_Kolléga_1_A” „Konk_Kolléga_3_B”, „Konk_Kolléga_2_A”, „Konk_Kolléga_4_B”*.
Speciális összetett döntésstruktúrával rendelkezik, mivel a közvetítő döntéshierarchia döntésesetét tartalmazzák az összehangolni kívánt *„Kolléga”* döntés hierarchiák döntésesetei.
Ugyanakkor a *„Közvetítő”* döntéshierarchia döntésesetei a *„Kolléga”* döntés döntéseseteit „beégetve” kezelik, azaz előre megadott *„Kolléga”* döntéslehetőségek közötti „közvetítést” szervezik. Kiemelt döntésszerkezet szerint ez azt jelenti, hogy a *„Közvetítő”* alosztályokban szereplő *„KonkrétKolléga”* objektumok referencia típusai nem szülő osztály, azaz *„Kolléga”* típusúak, hanem azok típusai ténylegesen a *„KonkrétKolléga”* osztályok. Ebben az irányban tehát a polimorfizmus nem érvényesül, ezért látnunk kell, hogy a *„Kolléga”* döntés esetén csak közös viselkedést szabunk meg, miszerint értesíteni kell változás esetén a *„Közvetítőt”*. Az eset szempontjából nem lényeges, hogy milyen egyéb elvek szerint volt szükséges kiemelni a döntést osztályhierarchiába.
Összegezve: az egyik döntéshierarchia (*„Kolléga”*) több döntésesetét egy másik döntés (*„Közvetítő”*) döntésesetén keresztül szinkronizáljuk. A *„Közvetítő”* döntéslehetőségei (*„KonkrétKözvetítő”*) előre meghatározott *„KonkrétKolléga”* döntésesetek (visszafelé irányban nincs *„Kolléga”* döntés!) együttműködését biztosítják. Tehát a *„Közvetítő”* döntéshierarchia azt a döntést tartalmazza, hogy mely *„KonkrétKollégák”* közötti együttműködést szándékozik megoldani és azt hogyan.
A „Közvetítő” tervezési minta célja, hogy egy döntéshierarchia – *„Kolléga”* döntéseseteit képes legyen összekötni egy olyan „közvetítővel”, amely a *„Kolléga”* egyedek előre meghatározott halmazainak elemei közötti együttműködést biztosítandó *„Kolléga”* halmazonként külön-külön döntéslehetőséget vezet be.

5.1.3.7 Megfigyelő tervezési minta

Objektumok közötti egy-sok kapcsolat esetén használatos, amikor egy adott objektum állapota központi szereplőként kihat a hozzá „tartozó” objektumok állapotára, viselkedésére, ezért azokat az állapotváltozásról értesíteni kell.
A Smalltalk megjelenésével vált ismerté az MVC (Modell-View-Controller) felépítés, működési megoldás, amely egyik alap tervezési mintája a „*Megfigyelő*”. Hiszen az MVC modell esetén, ha változik a modell, azaz a programállapot, akkor egy kontroller irányításával a szükséges felületi elem állapotváltozásokat végre kell hajtani.
A „*Közvetítő*” mintához hasonlóan itt is két felület határozza meg a szerepköröket. Az egyik szerepkör a „*Megfigyelő*”, amelyeket „regisztrálni” kell egy „*Alany*” egyeden, amely után az „*Alany*” objektum állapotának változása esetén a regisztrált „*Megfigyelő*” objektumok értesülnek az állapotváltozásról, és a szükséges állapotváltozások a „*Megfigyelő*” objektumokon is megtörténnek.



29. Diagram: Megfigyelő tervezési minta osztálydiagram, döntésszerkezet aktivitás diagram

Döntésszerkezet szerinti jellemzés:
Összetett döntésstruktúra, ahol – a *„Közvetítő”* tervezési mintához hasonlóan – két döntéshierarchia több döntésesetének együttműködését kell megszervezni: *„Alany”*

- *„Alany_1”* döntéshierarchia, amely beégetett döntéslehetőségei, mint „változásmenedzser” döntésesetek jelennek meg: *„K_Alany_1_A_Szerez”, „K_Alany_1_B_Szerez”,* illetve a *„Megfigyelő” – „Megf_Frissít_1”, „Megf_Frissít_2”* döntéshierarchia és döntésesetek.
Speciális összetett döntésstruktúrával rendelkezik, mivel a célt, azaz összehangolni kívánt döntés hierarchiák döntéseseteit (*„Megf_Frissít_1”, „Megf_Frissít_2”*) tartalmazzák a változásmenedzser/közvetítő – *„Alany”* döntéshierarchia döntésesetei („*Alany_1*”).
Ugyanakkor a *„Megfigyelő”* döntéshierarchia döntésesetei (*„Megf_Frissít_1”, „Megf_Frissít_2”*) az *„Alany”* döntés döntésesetét „beégetve” kezelik (*„K_Alany_1_A_Szerez”, „K_Alany_1_B_Szerez”*), azaz előre megadott *„Alany”* döntéslehetőség szerinti állapotváltozásra reagálnak csak. Kiemelt döntésszerkezet szerint ez azt jelenti, hogy a *„Megfigyelő”* alosztályokban szereplő *„KonkrétAlany”* objektumok referencia típusai nem szülő osztály, azaz *„Alany”* típusúak, hanem azok típusai ténylegesen a *„KonkrétAlany”* osztályok. Ebben az irányban tehát a polimorfizmus nem érvényesül, ezért látnunk kell, hogy az *„Alany”* döntés esetén csak közös viselkedést szabjuk meg, illetve azt az állapotot, amely változását a „megfigyelőknek” követni kell. Ebből a szempontból nem lényeges, hogy milyen egyéb elvek szerint volt szükséges az *„Alany”* döntéshierarchiánál a kiemelt döntés esetén az osztályhierarchiába szervezés.

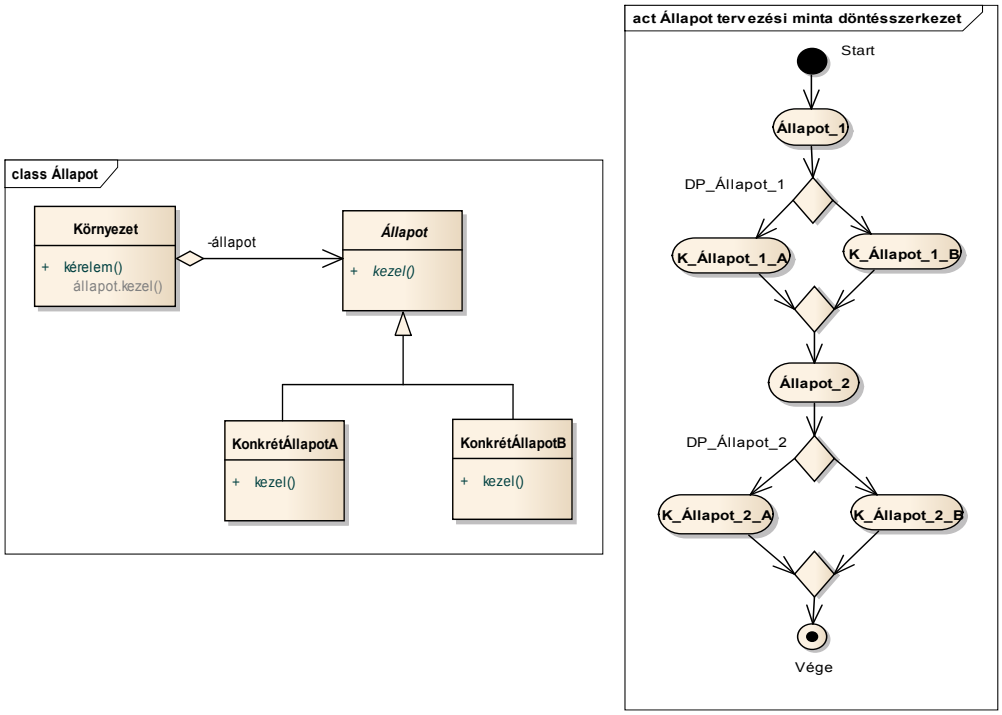
Összegezve az egyik döntéshierarchia („*Megfigyelő”*) több döntésesetéhez egy másik döntés („*Alany”*) ugyanazon döntésesetét kapcsoljuk, amely egyes döntéslehetőségnek („*KonkrétAlany”*) megfelelő döntésesetei több *„Megfigyelő”* döntésesetet értesítenek az esetleges állapotváltozásokról.
A *„Megfigyelő”* döntéslehetőségek előre meghatározott *„KonkrétAlany”* szerinti értesítést fogadnak csak el, vagyis előre meghatározott döntéseset szerinti döntéslehetőség – *„KonkrétAlany”* állapotváltozásának eseményére reagálnak csak.
Tartalmaz egy változás menedzser döntéshierarchiát –*„Alany”*, amely döntés lehetősége határozza meg az adott állapotot, melynek változását a regisztrált megfigyelői követni akarják, illetve amely állapot az adott esetben a megfigyelők számára lényeges. Látható tehát, hogy az *„Alany”* döntéshierarchia egyes döntéslehetőségei az állapot struktúrában különböznek, amitől a *„Megfigyelők”* valamilyen tulajdonsága függ, ezért szükségük van arra, hogy arról tudomást szerezzenek.

A *„Megfigyelő”* tervezési minta a *„Közvetítő”* tervezési mintához hasonlóan egy döntéshierarchia egy döntésesetén keresztül szinkronizálja a másik döntéshierarchia több döntésesetének állapotait. De eltérő megoldásokat adnak. A *„Közvetítő”* tervezési minta esetén a *„Közvetítő”* döntéshierarchia döntésesetei az összehangolni kívánt „*Kolléga*” döntés döntéseseteit „beégetve” kezelik, azaz előre megadott *„Kolléga”* döntéslehetőségek közötti „közvetítést” szervezik.
Ezzel szemben a *„Megfigyelő”* tervezési minta esetén az állapotváltozás szerint összehangolni kívánt *„Megfigyelő”* döntésesetek döntéslehetőség meghatározása dinamikusan történik. A *„Megfigyelő”* tervezési minta esetén a *„Megfigyelő”*

döntéshierarchia döntéslehetőségeibe vannak „beégetve” azok a „*KonkrétAlany”* döntéslehetőségek, amelyek szerinti állapotváltozást követni kell.
Az alapján kell eldönteni, hogy a két megoldás közül melyiket kell használni, hogy az összehangolni kívánt komponensek előre meghatározottak – „*Közvetítő”* tervezési minta, vagy az összehangolni kívánt komponenseket készítjük fel arra, hogy azokat „ki akarja összehangolni” – „*Megfigyelő”*.

5.1.3.8 Állapot tervezési minta

Egy objektum számára állapotfüggő viselkedést biztosít úgy, hogy ha egy objektumnak megváltozik az állapota, akkor az új állapotnak megfelelő viselkedést rendel hozzá.



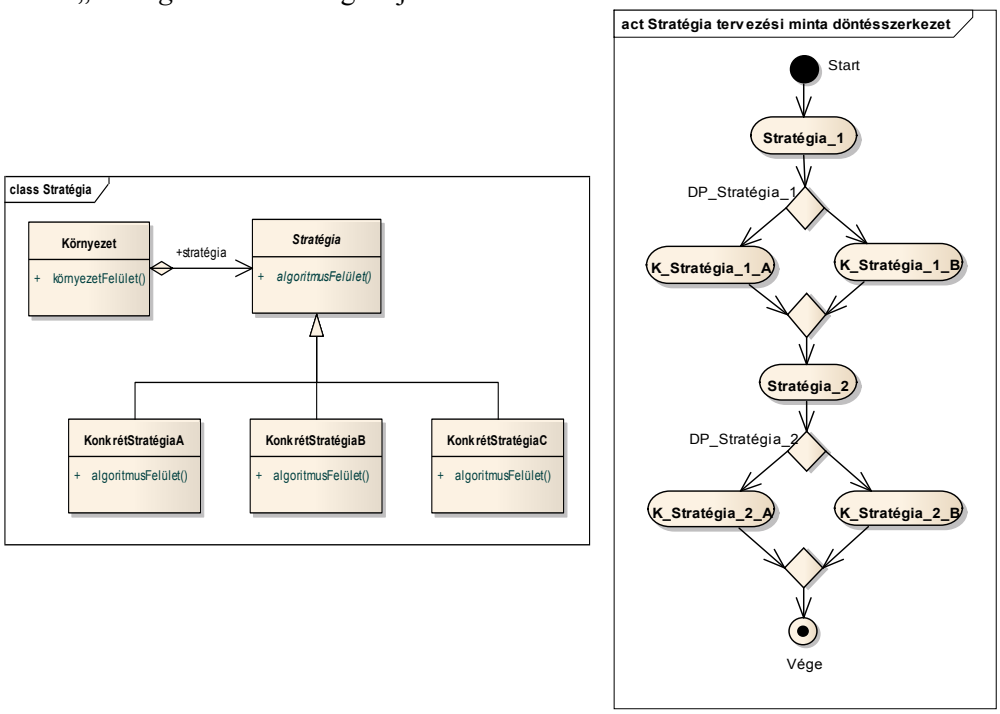
30. Diagram: Állapot tervezési minta osztálydiagram, döntésszerkezet aktivitás diagram

Döntésszerkezet szerinti jellemzés:
„Állapot” tervezési minta esetén viselkedésszerződésileg ekvivalens döntésekkel (*„Állapot_1”,”Állapot_2”*) találkozunk (döntéspredikátumok ekvivalenciája nem követelmény), melyek döntéspredikátumainak (*„DP_Állapot_1”, „DP_Állapot_2”*) meghatározásában részt vesz a döntések belső adatszerkezete, azaz a döntéslehetőségek állapotleírását meghatározó adatszerkezettől függ a döntéspredikátum kiértékelése, azaz a megfelelő döntéslehetőség kiválasztása
Egy adott komponens állapotfüggő viselkedését egy döntéshierarchia szolgáltatja úgy, hogy a különböző állapotok szerinti viselkedést az adott döntéshierarchia döntéslehetőségei adják. Kiemelt döntés esetén az állapot szerinti döntéslehetőségeket a döntéshez létrehozott felület alosztályaiként deklaráljuk,

aminek segítségével egy *„Állapot”* típusú referenciával hivatkozva valamelyik döntéslehetőség szerinti alosztály példányt, a döntést bezárva a későbbi döntésesetek helyein (ha a döntéspredikátumuk is ekvivalens volt a kiemelt és összevont döntéseknek) felhasználhatjuk.
Összegezve: egy környezet állapotfüggő viselkedéseiről döntünk, amely lehetőségei egy döntés döntéslehetőségeiként jelennek meg.

5.1.3.9 Stratégia tervezési minta

Egy objektum (*„Környezet”* egyede) számára, ha a viselkedés egy részletét cserélhetővé akarjuk tenni, akkor a *„Stratégia”* minta lehetőséget ad arra, hogy a *„Stratégia”* felületen keresztül az alosztályokban különböző megoldásokat adjunk, amit a *„Stratégia”* felület mögé rejtve használhatunk fel.



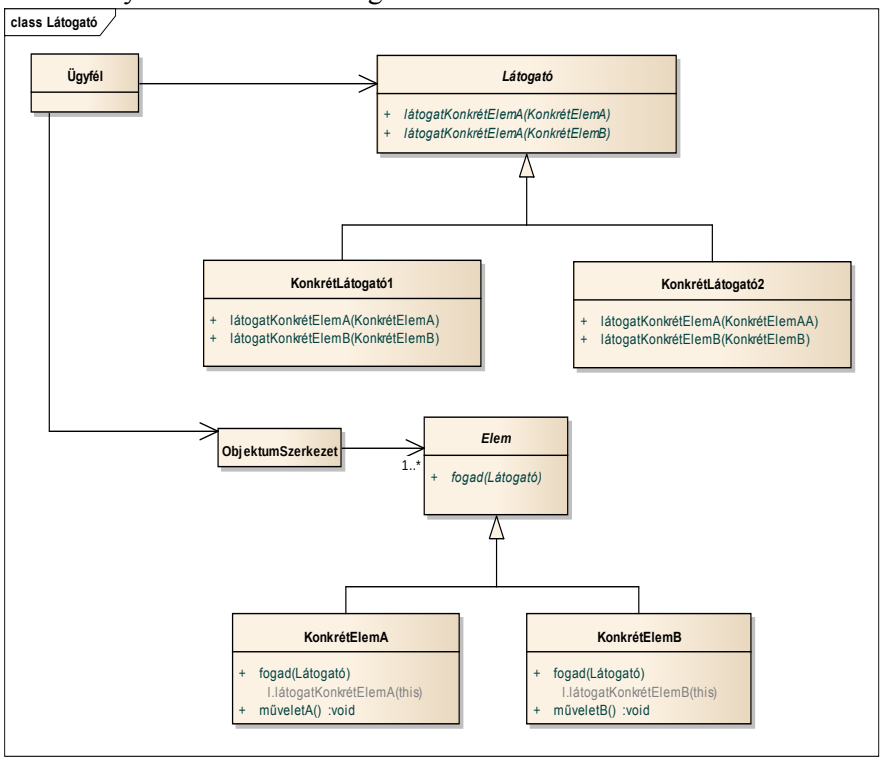
31. Diagram: Stratégia tervezési minta osztálydiagram, döntésszerkezet aktivitás diagram

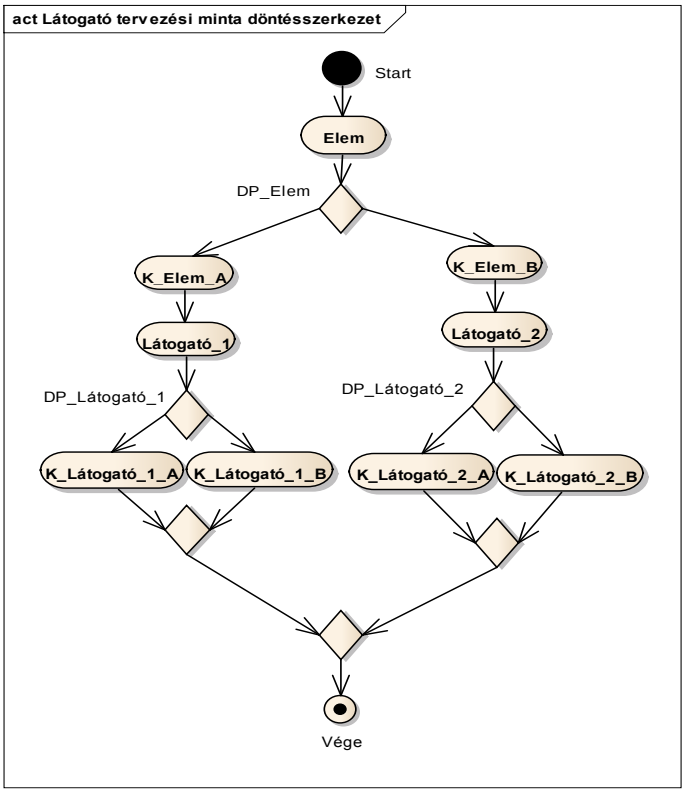
Döntésszerkezet szerinti jellemzés:
„Stratégia” tervezési minta esetén viselkedésszerződésileg ekvivalens döntésekkel *(„Stratégia_1”,”Stratégia_2”)* találkozunk (döntéspredikátumok ekvivalenciája nem követelmény), amelyek kiemelhetők és összevonhatók egy döntéshierarchiába lehetővé téve a döntéslehetőségek szerinti viselkedések egy helyen történő deklarálását, illetve a döntéspredikátum ekvivalencia (DP_Stratégia_1 = DP_Stratégia_2) teljesülése esetén értelmet nyerő döntésarchiválást.
A program egy adott pontján a viselkedést, megfelelő algoritmust egy döntés döntéslehetőségei szolgáltatják úgy, hogy a különböző algoritmusokat az egyes döntéslehetőségek képviselik. Dönteni kell arról, hogy a lehetőségek közül melyik

algoritmust használjuk fel, és a döntés szerint választott algoritmus döntéslehetőségét bezárva a későbbi döntésesetek helyein újra felhasználható.
Az „*Állapot”* és „*Stratégia”* tervezési minták hasonló elven működnek. A különbség csupán az, amíg az „Állapot” tervezési minta esetén a nevéből is következően állapotfüggő viselkedéssel kapcsolatos döntéseket kezelünk, a „*Stratégia”* tervezési minta egy algoritmus viselkedésének cserélhetőségét kezeli. Döntésszerkezetileg a két tervezési minta között nincs különbség.

5.1.3.10 Látogató tervezési minta

A „*Látogató”* tervezési minta segítségével rugalmasan lehet objektumszerkezeteken értelmezett műveleteket kezelni. Új művelet bevezetése az adott objektumokon úgy lehetséges, hogy nem szükséges osztályaikon módosítani.
A „*Látogató”* tervezési minta egy elvont „*Látogató”* felület alosztályaival biztosítja az objektumszerkezet osztályainak egyedei számára a szükséges műveleteket, amelyek új művelettel bővíthetők új alosztály bevezetésével. A „*Látogató”* felület az elvont „*Elem”* felület alosztályai számára nyújt különböző műveleteket úgy, hogy a „*Látogató”* felületben minden egyes „*Elem”* alosztály számára lennie kell arra az alosztályra vonatkozó műveletnek. Az egyes „*Elem”* alosztályokra vonatkozó műveleteket a „*Látogató”* felületen az alapján azonosíthatjuk, hogy az „*Elem*”„fogad” metódusának adott alosztály szerinti megvalósításában a „*Látogató”* felület mely metódusa kerül meghívásra.





32. Diagram: Látogató tervezési minta osztálydiagram, döntésszerkezet aktivitás diagram

Döntésszerkezet szerinti jellemzés:
Három döntéshierarchia összekapcsolásával jön létre, melynek célja, hogy két döntéshierarchia (*„Látogató_1”, „Látogató_2”*) a másik döntéshierarchia (*„Elem”*) döntésének megfelelően valamilyen viselkedést biztosítson.
A viselkedést szolgáltató *„Látogató_1”, „Látogató_2”* döntéshierarchiák döntéslehetőségei a nyújtandó viselkedést határozzák meg az *„Elem”* döntéshierarchia döntéslehetőségeinek. A *„Látogató_1”, „Látogató_2”* döntéshierarchiák viselkedésileg nem, de döntéspredikátumukban ekvivalensek (*DP_Látogató_1=DP_Látogató_2*), ezért a döntésösszevonás szabályai szerint összevonhatóak egy döntéshierarchiába, amellyel létrejövő *„Látogató”* osztályhierarchiával egyszerre tudják szolgáltatni a megfelelő viselkedést az „Elem” döntéshierarchia döntéslehetőségei számára.
Tehát a *„Látogató”* tervezési mintának olyan összetett döntésszerkezete van, ahol az *„Elem”* döntéshierarchia döntéslehetőségei tartalmazzák a műveletre vonatkozó döntéseket a *„Látogató_1”, „Látogató_2”* döntéshierarchiákon keresztül úgy, hogy azok döntéspredikátumai ekvivalensek.
Összegezve: ha egy döntés („Elem”) döntéslehetőségeiben olyan döntések (*„Látogató_1”, „Látogató_2”*) vannak, amelyek döntéspredikátumai ekvivalensek, de nem ugyanazon viselkedésről, illetve adatstruktúráról döntenek, akkor a tartalmazott, ekvivalens döntéspredikátumú döntések összevonhatók egy osztályhierarchiába úgy, hogy az képes a tartalmazó döntéslehetőségeknek

megfelelő metodológiára vagy adatstruktúrára irányuló tartalmazott döntéseket együtt kezelni. Tehát a döntéslehetőségből kiemelt döntések definícióját összevonjuk közös hierarchiába, és az alkalmas lesz a döntéslehetőségekben lévő döntések kiváltására figyelembe véve, hogy mely döntéslehetőségben kerül felhasználásra (metódus név konvenció).

5.1.4 Tervezési minták osztályozásának új módszere

A tervezési minták osztályozhatók aszerint, hogy milyen döntésszerkezet esetén adják meg az optimális döntésdeklarációkat, mellyel a döntésredundancia elkerülése biztosítható. A mintákat két csoportra oszthatjuk, amit külön osztályozhatunk.
Az egyik csoport a különböző döntésszerkezetek optimális struktúrájának kialakítására ad megoldásokat, a másik csoport a döntés archiválás különböző döntésszerkezetek szerinti megoldásait taglalja.
A tervezési minták döntésszerkezetük szerinti vizsgálatával felhasználásuk egy új irányát vehetjük észre, miszerint a döntések nem kiemelt állapota esetén (metódusba ágyazva deklaráltak) is észlelhetők azok a döntéskapcsolatok, amelyek az adott tervezési mintára jellemzőek, így a tervezési minták nem csak a jelenlegi formájukban ismerhetők fel, hanem a döntések nem kiemelt állapota esetén a tervezési minták „lenyomatai” is észlelhetők. Tehát a döntéskapcsolatok meghatározzák, hogy a döntésredundancia megszüntetése melyik tervezési minta szerinti struktúra felvételével valósulhat meg.
Ez új lehetőséget ad annak tisztázására, hogy milyen esetekben lehetséges egy adott tervezési minta felhasználása, hiszen egy tervezési minta jellemzésekor nem egy optimális megoldás paramétereit tudjuk így megadni, hanem egy olyan program strukturális helyzet körülményeit, amikor egy adott tervezési minta felhasználása indokolt lehet. Abban az esetben, ha a döntések elemzése automatizált eszközökkel is megtehető, akkor lehetőség nyílhat a tervezési minta alkalmazás szükségességének automatikus felismerésére.

A döntések egymásba épülése szerinti osztályozás:
A következőkben a döntésredundancia megszüntetés esetei és a döntések tulajdonságai alapján fogom a tervezési mintákat csoportosítani

- Ekvivalens döntéspredikátumú döntések, melyek nem egyező adattartalmat és metodológiát definiálnak, összevonhatók:

  „*Sablonfüggvény*”

  - Ha két döntésnek vannak olyan döntéslehetőségei, amelyek döntéspredikátuma ekvivalens és döntéslehetőségeik nem ugyanazon metodológiát és/vagy adatstruktúrát definiálják, de a fenti esetektől eltérően nem akarjuk összevonni őket egy osztályhierarchiába, mert két alapvetően eltérő „jellemvonást” határoznak meg, akkor lehetséges az, hogy a két döntés megfelelő döntéslehetőségeit együtt definiáljuk többszörös öröklődéssel:

    „*Osztályillesztő*”
- A döntések döntéslehetőségei ugyanazon adattartalmat, funkcionalitást definiálják, melyek kiemelhetők egy hierarchiába, amely az adott döntést

„bezárni”, archiválni képes (opcionálisan döntéspredikátumaik ekvivalensek):

„*Állapot*”, „*Stratégia*”, „*Parancs*”, „*Objektumillesztő*”

- Összetett döntések esetei: (Döntés tartalmaz döntést.)
  - Ha egy döntés döntéslehetőségeibe egyező viselkedésű döntéseket ágyazunk:

    „*Híd*”, „*Közvetítő*”
    - Ha egy döntés döntéslehetőségeibe több egyező viselkedésű döntést ágyazunk:

      „*Megfigyelő*”
  - Ha egy döntés olyan döntéseket tartalmaz, amely döntéspredikátuma ekvivalens a tartalmazott döntésével:

    „*Bejáró*”
  - Ha egy döntésbe nem ekvivalens döntéspredikátumú, de vele egyező viselkedésű döntést ágyazunk , vagyis a döntés minden döntéslehetőségébe vele megegyező, de a döntéspredikátummal nem ekvivalens döntést ágyazunk:

    „*Felelősséglánc*”
  - Ha a döntésnek az egyik döntéslehetőségébe ágyazunk az adott döntés döntéslehetőségének viselkedését kiegészítő döntést:

    „*Díszítő*”
  - Ha a döntésnek az egyik döntéslehetőségébe több az adott döntéssel viselkedés szerint egyező, de döntéspredikátumában nem ekvivalens döntést ágyazunk:

    „*Összetétel*”, „*Értelmező*”
  - Ha a döntésbe az adott döntéssel megegyező viselkedésű döntés egy döntéslehetőségét ágyazzuk:

    „*Helyettes*”
  - Ha egy döntés döntéslehetőségeiben olyan döntések vannak, amelyek döntéspredikátumai ekvivalensek, de nem ugyanazon viselkedésről illetve adatstruktúráról döntenek. A tartalmazott, ekvivalens döntéspredikátumú döntések összevonhatók egy osztályhierarchiába úgy, hogy az képes a tartalmazó döntéslehetőségeknek megfelelő metodológiára vagy adatstruktúrára irányuló tartalmazott döntéseket együtt kezelni. Tehát a döntéslehetőségből kiemelt döntések definícióját összevonjuk közös hierarchiába, és az alkalmas lesz a döntéslehetőségekben lévő döntések kiváltására, figyelembe véve, hogy mely döntéslehetőségben kerül felhasználásra.

    Kétszintű döntéshierarchia esetén kell tehát alkalmazni a „Látogató” tervezési mintát, ahol a tartalmazó döntés döntéslehetőségeiben olyan döntések vannak, melyek függenek a tartalmazó döntésektől, de a döntéspredikátumaik ekvivalensek:

    „*Látogató*”

Döntésarchiválás típusai szerinti osztályozás:
Abban az esetben, ha a döntés eredményét hatókörön kívül szeretnénk felhasználni mert a döntéshozatal és a felhasználás külön van választva, vagy a döntésnek megfelelően az egyes döntési pontokon új adattartalmat és metodológiát kell gyártani, akkor a döntésarchiváló tervezési minták adnak recepteket a probléma kezelésére.

Az archiváló tervezési mintákat a következő csoportokba sorolhatjuk:

- Nem a terméktípussal kapcsolatban hozunk döntést, hanem a gyártási mechanizmussal kapcsolatban.
- A termék típusával kapcsolatos döntésünket egy gyártó osztály hierarchiával archiváljuk, és később a tárolt döntéssel tudjuk legyártani a megfelelő termék egyedeket.
- A döntést úgynevezett döntésbélyeggel archiváljuk, és a döntéshelyeken a döntésbélyeggel archivált döntéshez „testet” egy úgynevezett döntéstárból kérhetünk.

A második esetet használva a termék típusa alapján dől el, hogy melyik megoldást kell használni. Az adja meg a döntésarchiválás módját, hogy a termékdöntések körülménye, azaz a döntéseknek megfelelő metodológiák és adatszerkezetek illetve a döntéspredikátumok milyen kapcsolatban vannak egymással. A jelenleg ismert „Létrehozási minták” nagy része ebbe a csoportba sorolható. Fontos megjegyezni, hogy a csoportosításban szerepelni fognak olyan tervezési minták is, amelyek viselkedési vagy szerkezeti minták, de jellemzőik alapján egyértelműen rendelkeznek olyan tulajdonságokkal, melyek alapján az itt megadott felsorolásba is meg kell, hogy jelenjenek. A fentebb megadott három csoport szerint osztályozva a tervezési mintákat a következő csoportokat kapjuk:

- Ha nem a gyártandó termékekről akarunk dönteni, hanem a gyártási mechanizmusról. A gyártandó termékek nem sorolhatók egy döntési hierarchiába:

  „*Építő*”
- A termék típusával kapcsolatos döntésünket egy gyártó osztály hierarchiával archiváljuk és később a tárolt döntéssel tudjuk legyártani a megfelelő termékegyedeket.
    - Ha egy termékkel kapcsolatos döntéshez akarunk döntésarchiválási támogatást adni, akkor egy „kiemelt” termék döntés gyártásához szükséges egy gyártó döntés hierarchia, aminek újrahasználhatósága érdekében azt is archiválni kell. Ebben az esetben tehát van egy gyártó és termék döntéshierarchia, amelyek egymással döntéspredikátum szerint ekvivalensek, döntéslelhetőségeik párba állíthatók, azaz minden termék döntéslehetőséghez van egy gyártó döntéslehetőség:

      „*Gyártó függvény*”, „*Bejáró*”
    - Ha az előző esethez hasonlóan egy termékkel kapcsolatos döntéshez akarunk döntésarchiválási támogatást adni, akkor egy termék döntés gyártásához szükséges egy gyártó döntés hierarchia, melyek döntéspredikátumai ekvivalensek. Korábban taglaltak

szerint ekvivalens döntéspredikátumú döntések összevonhatóak, azaz lehetséges a gyártó és termék hierarchia összevonása. (A gyártó és termék szerepkört összevonjuk.):

„*Prototípus*”

- Ha vannak olyan termékdöntések, amelyek döntéspredikátumai ekvivalensek, azaz mindig ugyanakkor teljesülnek, akkor a termékdöntés hierarchiákra közös gyártó hierarchiát készíthetünk, amely a kiemelt termék döntéshierarchiákra összevontan használható, azaz összevonva tudja archiválni a gyártási döntéseket, és gyártani az adott döntéseknek megfelelő termékdöntéseket. („*Gyártó függvény*” hierarchiák összevonása, ami akkor lehetséges, ha a „gyártó” döntések döntéspredikátumai ekvivalensek.):

  „*Elvont gyár*”

- A döntéstárban tárolt döntéseredmények, döntésbélyeggel archivált döntések újrafelhasználása:
  - Ha ekvivalens döntéspredikátumú döntéseket úgy vonjuk össze, hogy a döntés eredményét egy döntéstárba tesszük és az egyes döntési helyeken egy úgynevezett döntésbélyeg segítségével kérjük ki azokat a döntéstárból:

    „*Pehelysúlyú*”

5.2 Sablonfüggvény tervezési minta döntésszerkezet szerinti felismerése az „eFilter” projekten

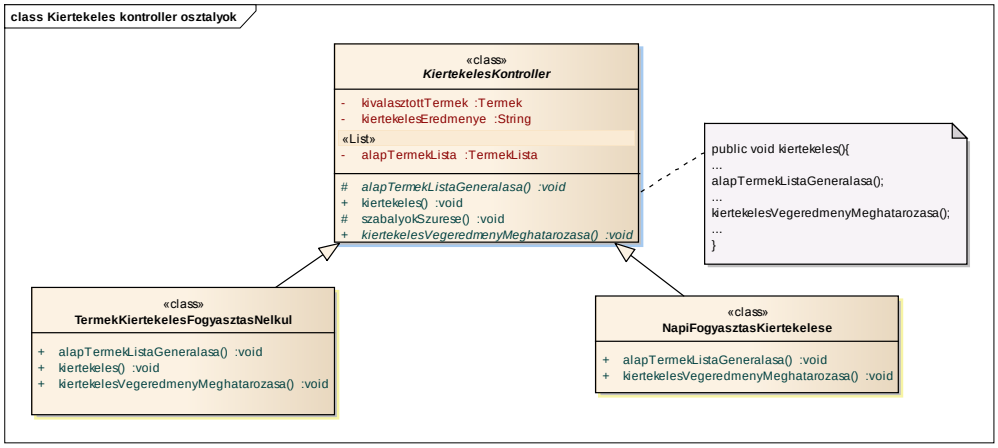
Az „eFilter” – „Egészségügyi profil alapján szűrt fogyasztói adatbázisokból nyert információkat kezelő rendszer” kutatásfejlesztési projekt [22] alkalmat kínált arra, hogy a felvázolt elképzelést, a tervezési minták döntésszerkezet alapján történő felismerésének lehetőségeit gyakorlatban is kipróbáljuk. Projektvezetőként és vezető tervezőként lehetőségem volt arra, hogy ráirányítsam a figyelmet az elemzés-tervezés során a tervezési minták és döntésszerkezetek közötti összefüggések kérdésére.

Sikerült több tervezési minta alkalmazhatóságát döntésszerkezet szerint észlelni, amelyek közül az egyik esetet a fejezetben ismertetek.

Meg kell jegyeznem, hogy a létrehozási tervezési minták nem jutottak szerephez, mivel lehetőségüket a használt „JBoss Seam” [19] fejlesztési keretrendszer használata „elfedte”.

Az „eFilter” rendszer modellezése során többször használtuk az „*Összetétel*” tervezési mintát annak köszönhetően, hogy a fa adatszerkezetek kezelése a rendszerben gyakori. A „*Sablonfüggvény*” viselkedési tervezési minta is felhasználásra került a kontroller modellben.

A „eFilter” projektben megjelenő „*Sablonfüggvény*” tervezési minta osztályszerkezete:

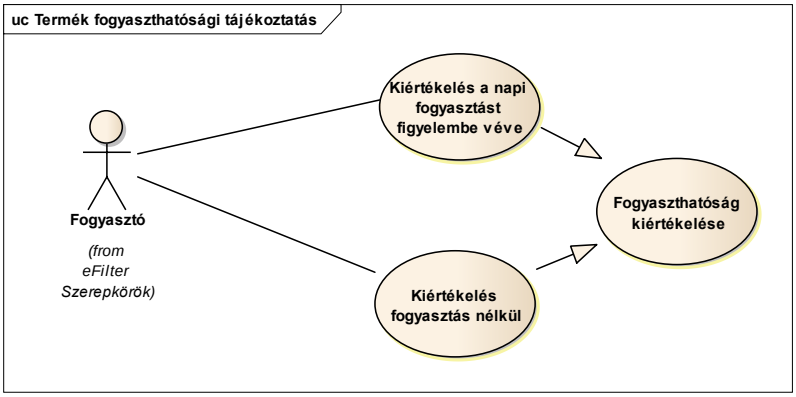


33. Diagram: Sablon függvény osztályszerkezet példa

Az „elemi műveletek” felületeit az „*alapTermekListaGeneralasa()*” és a „*kiertekelesVegeredmenyMeghatarozasa()*” absztrakt metódusok adják, melyek megvalósításait az alosztályokban találjuk. A „*kiertekeles()*” metódus a „sablonfüggvény”, amely elemi műveletekként használja fel az „*alapTermekListaGeneralasa()*” és a „*kiertekelesVegeredmenyMeghatarozasa()*” absztrakt metódusok felületét.

A minta használatának beazonosítása a kiértékelés funkció algoritmusának modellezésekor történt, ami közben észleltük azt, hogy a „kiértékelés” metódus tartalmaz felhasználói döntés alapján eltérő viselkedésű részeket. Hiszen felhasználói döntés alapján kerül meghatározásra az „elemi művelet” megvalósítása.

A „felhasználói döntés” már használati eset modellezésnél megjelent, ami előrevetítette a tervezési minta felhasználásának lehetőségét:



34. Diagram: Kiértékelés használati eset diagram

A használati eset diagramon lévő döntés két ekvivalens döntéspredikátumú döntésben „érvényesül”. Az egyik az alap termék lista generálás, ami az első döntéslehetőség szerint figyelembe veszi az adott nap korábbi fogyasztásait, azonban a másik esetben nincs szükség a napi fogyasztásokra. A másik döntés a kiértékelés végeredmény meghatározása, ahol az egyik esetben egy egyértelmű,

aggregált fogyaszthatósággal kapcsolatos eredmény megjelenítésére van szükség, amíg a napi fogyasztás figyelembe vétele esetén az összes egészségügyi profil szabálysértést meg kell jeleníteni.

5.3 Tervezési minták jellemzése döntéskiemelés előtt és után

Ha döntéskapcsolatok szerint vizsgáljuk a tervezési mintákat, akkor lehetőség van arra, hogy a tervezési mintákat ne csak a jelenleg ismert osztály és objektum szerkezetük alapján ismerjük fel, hanem a döntések egymásba épülése, azaz döntésszerkezetük alapján. Ezen a módon úgynevezett rejtett tervezési mintákat is felderíthetünk. Ha egy döntés még nem kiemelt állapotban van, a tervezési minta „lenyomata" a döntések struktúrája alapján észlelhető. Döntéskiemelésekkel az optimális struktúra kialakítható, ahol természetesen a döntés struktúra ugyanaz marad, csak a döntések deklarálása már strukturáltabb formában történik.

Ezek alapján a tervezési mintáknak két állapotát különböztetjük meg:

1. Döntéskiemelés előtt: A döntések és azok lehetőségei metódusokban vannak deklarálva.
2. Döntéskiemelés után: A döntések és azok döntéslehetőségei osztályhierarchiába definiáltak, a megfelelő döntésredundanciákat feloldva.

A tervezési minták detektálása szempontjából mindegy, hogy az adott döntések melyik állapotukban definiáltak, azaz ki vannak-e emelve osztály hierarchiába vagy még metódusban deklaráltak.
Erre alapozva lehetséges, hogy a tervezési minták detektálása döntéskapcsolatok szerint történjen. A nem kiemelt döntések esetén is észrevehetők azok a döntés-kapcsolatok, amelyek az egyes tervezési mintákra jellemzőek, melyek meghatározzák, hogy a döntésredundancia megszüntetése melyik tervezési minta szerinti struktúra felvételével valósulhat meg. A tervezési minták „lenyomata" a döntések struktúrája alapján észlelhető, tehát a „rejtett" tervezési minták felderítésére lehetséges.
A döntéskiemelés művelet végrehajtásával az optimális struktúra kialakítható, ahol természetesen a döntés struktúra ugyanaz marad, csak a döntések állapota kiemeltre változik.

Döntéskiemelés előtti tervezési minták jellemzése, leírása:
A döntések metódusokban lévő feltételes utasításokkal deklaráltak, ahol a tervezési mintánként különbözőképpen beágyazott döntések viselkedése és állapotleírása az adott feltételes utasítás különböző blokkjaiban vagy azokból hívott metódusokban van deklarálva.

Döntéskiemelés utáni tervezési minták jellemzése, leírása:
A döntéskiemelés utáni tervezési mintákban lévő döntések osztályszerkezetekben deklaráltak, amelyek az egyes mintáknál megadott kapcsolatban vannak egymással.

A tervezési minták jellemzése szerkezettől függetlenül csupán a döntésszerkezet alapján is megtehető. Továbbá szerkezetfüggő jellemzőkkel rendelkeznek attól függően, hogy a döntések már osztályszerkezetbe kiemelten vagy még beágyazva, nem kiemelten deklaráltak.
Az „5.1 Tervezési minták döntésszerkezetek szerinti értelmezése” fejezetben számos tervezési minta esetén megadtam, hogy milyen döntésszerkezettel rendelkeznek.

Az alábbiakban egy tervezési mintát először szerkezet függetlenül, döntésstruktúra alapján ismertetek, majd nem kiemelt állapot szerinti szerkezeti jellemzőit adom meg, amit a kiemelt állapotú szerkezetleírás követ. Ez alapján jól látható, hogy miként jelenhet meg egy tervezési minta nem kiemelt állapotban.

5.3.1 Híd tervezési minta döntéskiemelés előtt és után

A „*Híd”* tervezési minta döntések deklarációjának állapotától, típusától független jellemzője:

> Összetett döntésstruktúra, két döntéshierarchia dinamikus összekapcsolását jelenti, ahol az egyik döntéshierarchia valamilyen szolgáltatást biztosít a másiknak. A *Híd* tervezési minta esetén olyan összetett döntésszerkezetről van szó, ahol a tartalmazó döntéshez egy tartalmazott, szolgáltató döntést rendelünk.
> A „*Híd”* tervezési minta esetén egy döntés nem döntés specifikus része tartalmaz egy másik döntést, amelynek viszont csak döntés specifikus része van.

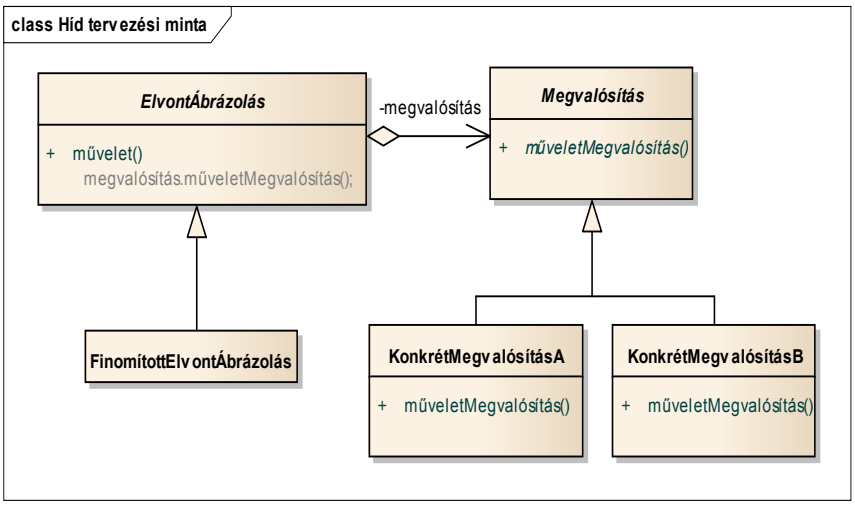
Híd tervezési minta megjelenítése nem kiemelt állapotban:
Nem kiemelt állapotban az „*ElvontÁbrázolás”* művelet metódusa és annak döntéslehetőségek szerinti változatai egy feltételes utasítás elágazásai szerint deklaráltak.
A „*Megvalósítás”* szerinti döntést be kell ágyazni az „*Elvont ábrázolás”* döntéslehetőségeibe:

```
if(„Elvont ábrázolás” döntéspredikátum){
        „Elvont ábrázolás” művelet 1. döntéslehetőség szerinti utasításai …

        if(„Megvalósítás” döntéspredikátum){
                „Megvalósítás művelet 1. döntéslehetőség szerinti utasításai …
        } else if(!„Megvalósítás” döntéspredikátum) {
                „Megvalósítás művelet 2. döntéslehetőség szerinti utasításai …
        }
} else if(!„Elvont ábrázolás” döntéspredikátum) {
        „Elvont ábrázolás” művelet 2. döntéslehetőség szerinti utasításai…

        if(„Megvalósítás döntéspredikátum){
```

```
            „Megvalósítás” művelet 1. döntéslehetőség szerinti utasításai …
        } else if(!„Megvalósítás” döntéspredikátum) {
            „Megvalósítás” művelet 2. döntéslehetőség szerinti utasításai …
        }
}
```

Meg kell jegyezni, hogy lehetőség van arra, hogy a beágyazott döntés deklarációja külön metódusba szervezetten jelenjen meg, amit most nem részletezek.

Híd tervezési minta megjelentése kiemelt állapotban:
Kiemelt állapotban az *„ElvontÁbrázolás”* művelet metódusa és annak döntéslehetőségek szerinti változatai egy osztályban és annak alosztályaiban deklaráltak.
Ebben az esetben a tartalmazó döntést reprezentáló *„ElvontÁbrázolás”* osztályhierarchia szülő osztályának lesz egy *„Megvalósítás”* típusú referenciája, amely objektum összetétellel a másik döntést reprezentáló osztályszerkezet absztrakt osztályára mutat. A tartalmazott *„Megvalósítás”* döntés ezen felületen keresztül érhető el, bezárva a tartalmazott döntést. Felhasználása a felület által elérhető absztrakt *„műveletMegvalósítás()”* polimorf metódus meghívásával lehetséges, amely a döntéslehetőség szerinti *„műveletMegvalósítás()”* implementáció meghívását jelenti. A tartalmazott döntés absztrakt osztálya adja meg tehát azt a felületet, ami a döntésnek keretet ad. A döntéslehetőségeket az alosztályok és az általuk implementált absztrakt metódus adja.



35. Diagram: Híd tervezési minta osztálydiagramja [15]

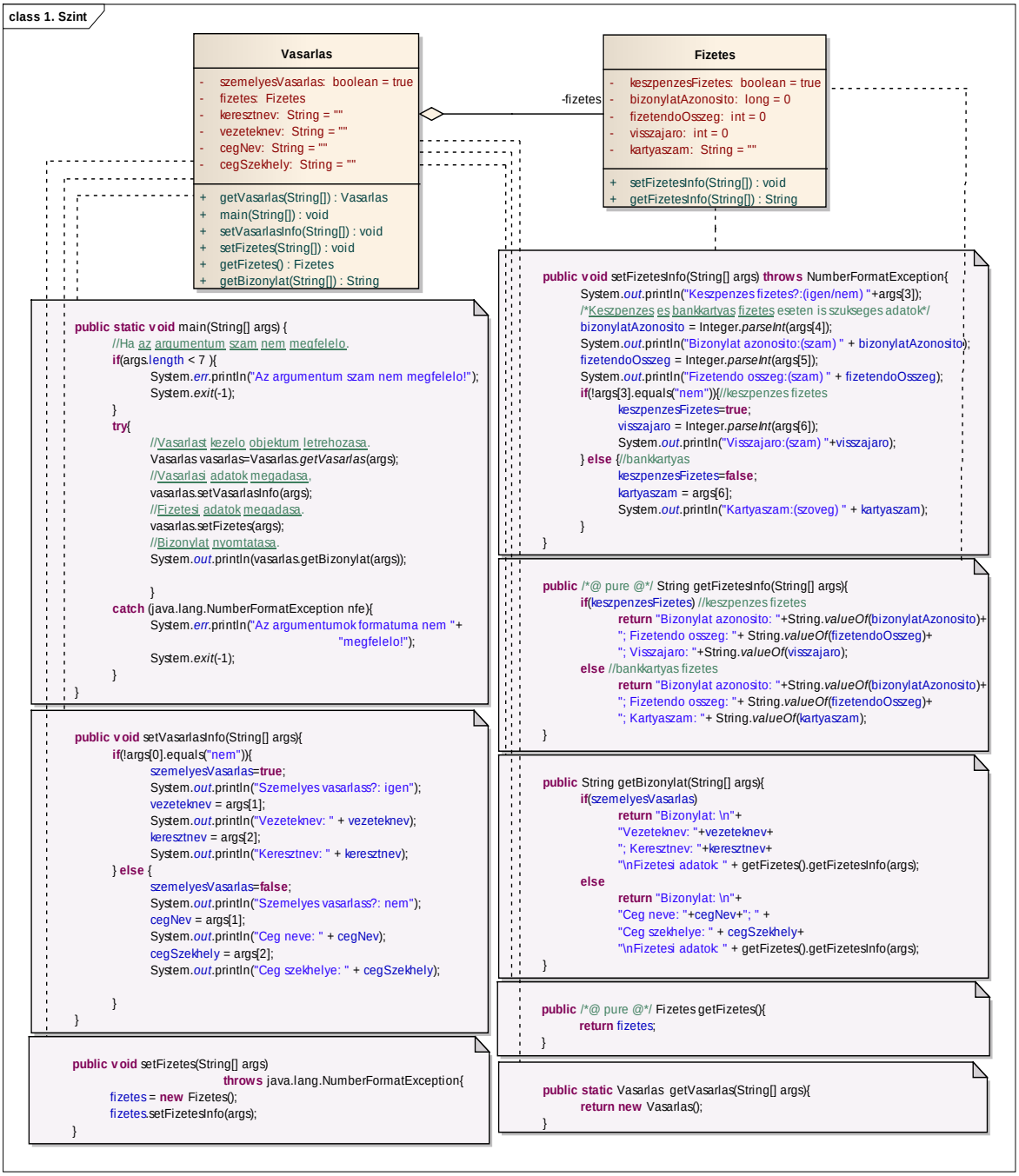
5.3.2 Példa – Híd tervezési minta döntéskiemelés előtt és után

A példaprogram egy személyes vagy céges vásárlást valósít meg, ahol fizethetünk készpénzzel vagy bankkártyával.

A program első döntése arra vonatkozik, hogy személyes vagy céges vásárlóról van-e szó. Ennek megfelelően más-más vásárlói információkra van szükség (1. döntés).
A második döntés alapján meghatározzuk, hogy a fizetés módja készpénzes vagy bankkártyás, ami alapján más-más fizetési adatok bekérése szükséges.
A hátralévő két döntés az előző döntések döntéspredikátumával ekvivalens döntések, azaz döntésismétlődések. Ezek szerint a vásárlás típusa alapján más-más adattartalommal kell a bizonylatot nyomtatni, ahol a fizetés módja alapján más-más fizetési adatokat kell feltűntetni.
Jelenleg a példaprogramban tehát ekvivalens döntéspredikátumú döntéseink vannak, amelyeket meg kell szűntetni. A „3.2 A döntésredundanciák elkerülésének esetei" fejezetben leírtak szerint az ekvivalens döntéspredikátummal rendelkező döntéseket össze kell vonni és ki kell emelni egy közös osztályhierarchiába, hogy ezzel a döntésredundanciákat megszűntessük.
A megszüntetés módja tehát a két ekvivalens döntés pár kiemelése egy-egy osztályhierarchiába.
A 4 döntésből 2-2 döntéspárt határoztunk meg, amelyek összevonhatók és kiemelhetők 1-1 osztályhierarchiába. Az így létrejött 2 döntéshierarchia kapcsolatban van egymással, mivel a vásárláskor kinyomtatott bizonylatnak (*Vasarlas.getBizonylat()*) tartalmaznia kell a vásárlói adatokon kívül a fizetési adatokat is (*Vasarlas.getFizetesInfo()*). Tehát az egyik döntés nem döntés specifikus része tartalmazza a másik döntést, amelynek csak döntés specifikus része van.
A példa döntésszerkezete a „*Híd*" tervezési minta döntésszerkezetével egyezik, vagyis a „*Híd*" tervezési minta alkalmazásával egy olyan optimálisabb osztályszerkezethez juthatunk, amely a döntésredundanciák megszüntetését eredményezi.
A döntésredundancia elkerülés, feloldás szabályainak („3.2 A döntésredundanciák elkerülésének esetei") alkalmazásával is eljuthatunk ahhoz az osztályszerkezethez, amely a „*Híd*" tervezési minta ismert megjelenése. Ezt több lépésben végezzük el, amivel egyben azt is szemléltetjük, hogy a „*Híd*" tervezési mintának milyen „rejtett" megjelenési formái lehetnek.

Összegezve tehát, ha jelen esetben a „rejtett" „*Híd*" tervezési mintát bevezetjük, megszüntetjük a kódban lévő döntésismétlődéseket, amit a döntéskiemeléssel, vagyis a döntések absztraktabb formára hozásával érhetünk el. A „*Híd*" tervezési minta azt a felépítést mutatja meg, miként kell az adott döntés kapcsolatok esetén a döntésismétlődést megszüntető megoldást felépíteni.
Fontos látnunk, hogy a tervezési minta adott ponton való alkalmazhatóságának lehetősége már nem kiemelt formában is látszik, hiszen a döntések kapcsolatai elárulják azt.
A példában Java szintaktikát [46] alkalmazok.

5.3.2.1 1. Szint – Döntésösszevonások és kiemelések előtti állapot



36. Diagram: Döntésismétlődéseket tartalmazó példaprogram osztálydiagram

A vásárlás típusával kapcsolatos döntések és a fizetés módjával kapcsolatos döntések nem kerültek még kiemelésre és összevonásra.

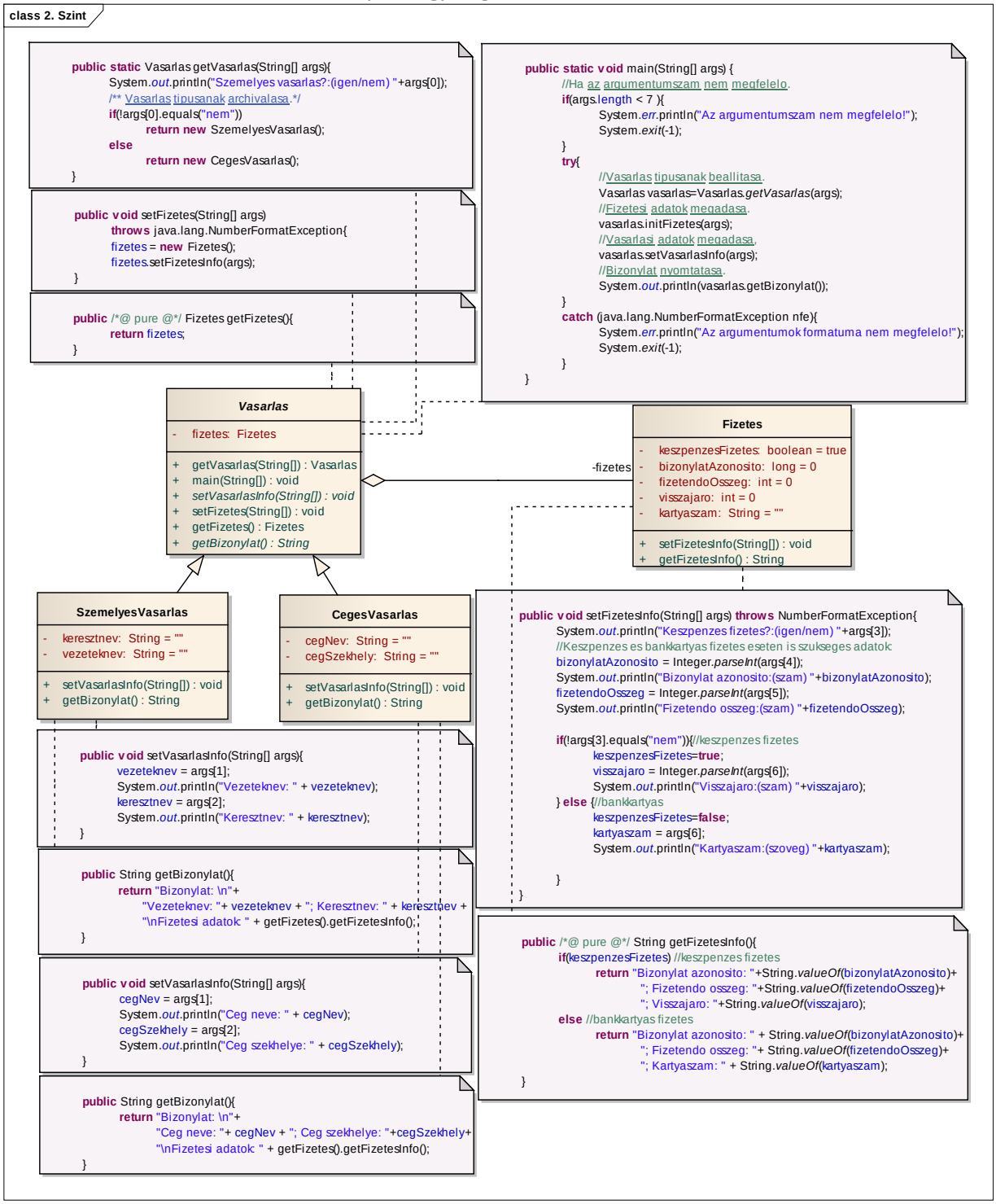
A vásárlással kapcsolatos döntésesetek, amelyek döntéspredikátumai ekvivalensek:

```
public void setVasarlasInfo(String[] args){
        if(!args[0].equals("nem")){
                szemelyesVasarlas=true;
                …
        } else {
                szemelyesVasarlas=false;
                …
        }
}

public String getBizonylat(String[] args){
        if(szemelyesVasarlas)
                …
        else
                …
}
```

A fizetés módjára vonatkozó ekvivalens döntéspredikátumú döntésesetek a következők:

```
public void setFizetesInfo(String[] args) throws
                                NumberFormatException{
        System.out.println("Keszpenzes fizetes?:(igen/nem) "
                                +args[1]);
        //Keszpenzes es bankkartyas fizetes eseten is szukseges
        adatok
        …
        if(!args[3].equals("nem")){//keszpenzes fizetes
                keszpenzesFizetes=true;
                …
        } else {//bankkartyas
                keszpenzesFizetes=false;
                …
        }
}

public /*@ pure @*/ String getFizetesInfo(String[] args){
        if(keszpenzesFizetes) //keszpenzes fizetes
                …
        else //bankkartyas fizetes
                …
}
```

5.3.2.2 2. Szint – „Személyes vagy céges vásárlás?” döntés kiemelése



37. Diagram: Első döntéskiemelés utáni példaprogram

A vásárlással (személyes vagy céges) kapcsolatos ekvivalens döntéspredikátumú döntéseket vontuk össze egy döntéshierarchiába, és „kiemelve” egy osztály hierarchiába absztraktabb formába deklaráltuk. Ezek után a vásárlással kapcsolatos döntések kezelése a következő módon jelenik meg a programban:
A *Vasarlas.getVasarlas()* metódusban egy *Vasarlas* típusú referenciával tároljuk, archiváljuk a döntést, amely egy *SzemelyesVasarlas* vagy egy *CegesVasarlas* típusú objektumra mutat:

```
            /** Vasarlas tipusanak archivalasa.*/
            if(!args[0].equals("nem"))
                  return new SzemelyesVasarlas();
            else
                  return new CegesVasarlas();
      }
```

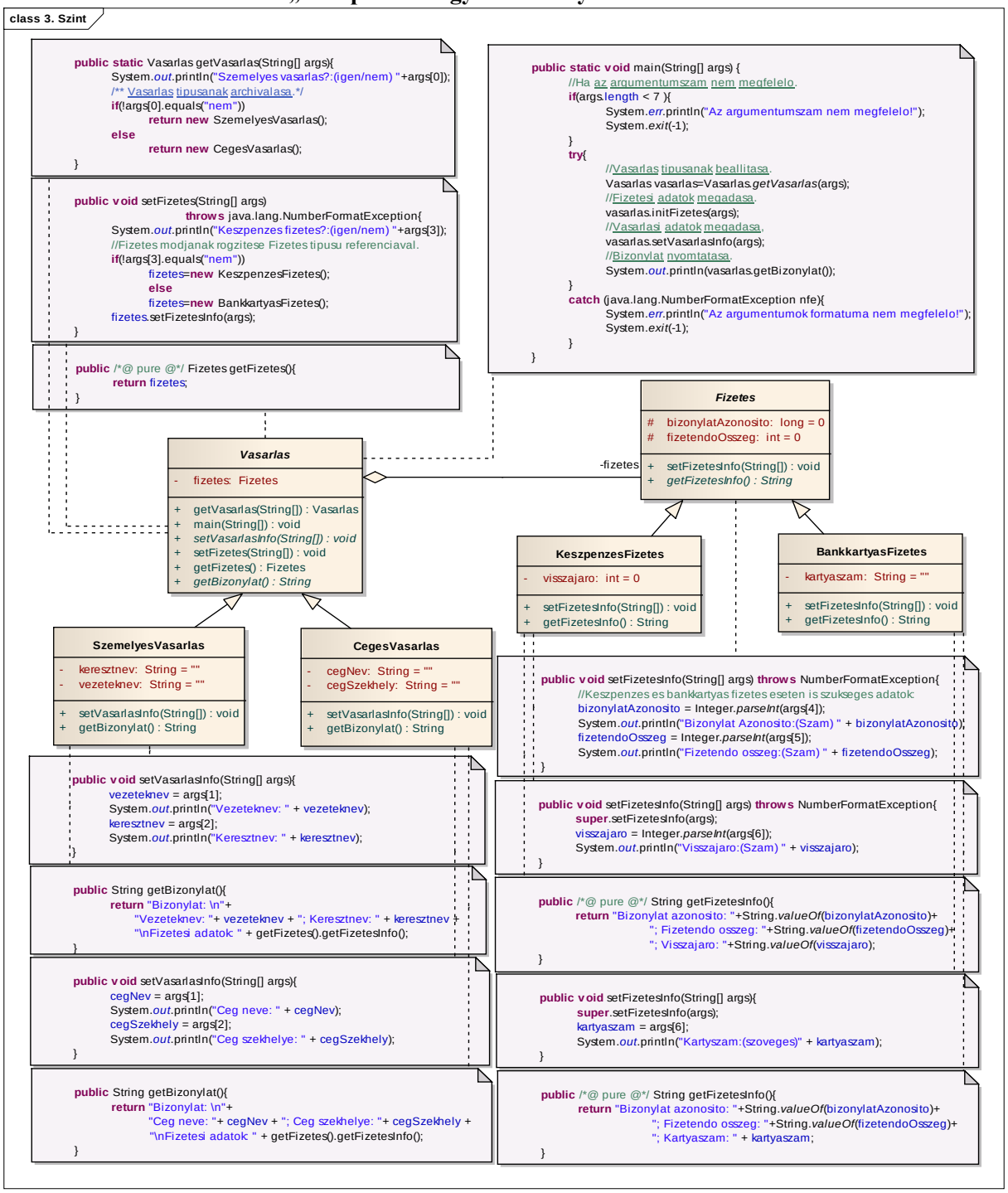
Az archivált döntés döntéspredikátuma: `!args[0].equals("nem")`, amely a későbbi döntési helyeken már nem jelenik meg
Az így archivált döntés felhasználási helyei a következők, ahol már rejtve marad, hogy melyik döntéslehetőség érvényesül:

```
            //Vasarlasi adatok megadasa,
            vasarlas.setVasarlasInfo(args);
            //Bizonylat nyomtatasa.
            System.out.println(vasarlas.getBizonylat());
```

Az első felhasználási hely a vásárlás típusfüggő vásárlási adatok bekérése. A második felhasználási hely a vásárlás típusfüggő bizonylat nyomtatása.
A példa előző szintjének bemutatásakor ez a két döntéseset külön-külön döntésként jelent meg, melyek döntéspredikátuma ekvivalens volt. Az összevonás után kezelésük már együtt történik.

A *SzemélyesVasarlas* osztály a *Vasarlas* osztály alosztályaként a döntésbezárás eszköze, amely akkor „érvényesül”, ha vásárlás típusára vonatkozó döntés értelmében személyes vásárlásról van szó. Személyes vásárlás esetén szükséges a vásárló vezeték és keresztnevének megadása, melyeknek rá kell kerülnie a nyomtatott bizonylatra is.
A *CegesVasarlas* osztály a *Vasarlas* osztály alosztályaként a döntésbezárás eszköze, amely akkor „érvényesül”, ha vásárlás típusára vonatkozó döntés értelmében céges vásárlásról van szó. Ebben az esetben a vásárló cégnevének, székhelyének megadására és nyomtatására van szükség.

5.3.2.3 3. Szint – „Készpénzes vagy bankkártyás fizetés?” döntés kiemelése



38. Diagram: Döntéskiemelések utáni példaprogram

A fizetés módjával (készpénzes vagy bankkártyás) kapcsolatos ekvivalens döntéspredikátumú döntéseket vontunk össze egy döntéshierarchiába és „kiemelve” egy osztály hierarchiába (Fizetés osztály és alosztályai) absztraktabb formába deklaráltuk. Ezek után a fizetéssel kapcsolatos döntések kezelése a következő módon jelenik meg a programban:
A *Vasarlas.initFizetes()* metódusban egy *Fizetes* típusú referenciával tároljuk, archiváljuk a döntést, amely egy *KeszpenzesFizetes* vagy egy *BankkartyasFizetes* típusú objektumra mutat:

```
//Fizetes modjanak rogzitese Fizetes tipusu referenciaval.
      if(!args[3].equals("nem"))
            fizetes=new KeszpenzesFizetes();
      else
            fizetes=new BankkartyasFizetes();
```

Az archivált döntés döntéspredikátuma: `!args[3].equals("nem")`, amely a későbbi döntési helyeken már nem jelenik meg
Az így archivált döntés felhasználási helyei a következők, ahol már rejtve marad, hogy melyik döntéslehetőség érvényesül:

```
      fizetes.setFizetesInfo(args);
      fizetes.getFizetesInfo();
```

Az első felhasználási hely a fizetés módjától függő adatok bekérése. A második felhasználási hely a fizetés módjától függő bizonylat nyomtatása.
A példa előző szintjeinek bemutatásakor ez a két döntéseset külön-külön döntésként jelent meg, melyek döntéspredikátuma ekvivalens volt. Az összevonás után kezelésük már együtt történik.

A *KeszpenzesFizetes* osztály a *Fizetes* osztály alosztályaként a döntésbezárás eszköze, amely akkor „érvényesül”, ha a fizetés módjára vonatkozó döntés értelmében készpénzes fizetésről van szó. Készpénzes fizetés esetén szükséges a bizonylatazonosító és fizetendő összeg (nem döntés specifikus paraméterek) mellett a visszajáró megadása, melynek rá kell kerülnie a nyomtatott bizonylatra is.
A *BankkartyasFizetes* osztály a *Fizetes* osztály alosztályaként a döntésbezárás eszköze, amely akkor „érvényesül”, ha a fizetés módjára vonatkozó döntés értelmében bankkártyás fizetésről van szó. Ebben az esetben a bizonylatazonosító és fizetendő összeg (nem döntés specifikus paraméterek) mellett, bankkártyaszám megadására és nyomtatására van szükség.

Látnunk kell, hogy a 3. szint bevezetésével, azaz a 2-2 ekvivalens döntéspredikátumú döntésösszevonásával és osztályhierarchiába történő kiemelésével kialakult osztályszerkezet a „Híd” tervezési mintának megfelelő felépítést tükrözi.

5.4 Tervezési minták döntésalapú formalizálása JML-lel

A tervezési mintákkal kapcsolatban ismertetett új értelmezésre alapozva célom a tervezési minták döntésszerkezetük szerinti formális leírása. Ennek megfelelően a tervezési minták JML-lel történő formalizálásának alapja a döntés szerkezetekben lévő döntések formalizálása. Lényeges, hogy miként kapcsolódhatnak össze a döntésszerkezetek, illetve ezen kapcsolódások JML-es specifikációi hogyan jelennek meg. (Lásd: „4. Döntések vizsgálata JML-lel" fejezet)
Ahogy azt a „5.1 Tervezési minták döntésszerkezetek szerinti értelmezése" fejezetben bemutattam, az egyes minták jellemzőit a bennük lévő döntések tulajdonságai alapján tudjuk meghatározni. A JML specifikációval megfogalmazott döntésekkel kapcsolatos mintákra vonatkozó megkötéseket ezek alapján tudjuk értelmezni.

A következőkben bemutatom egy lehetőségét annak, hogy JML-re alapozva miként tudjuk döntésszerkezet szerint formalizálni a tervezési mintákat. Amint látható lesz, érdemes a tervezési mintában található döntésszerkezet deklarációk különböző állapotainak formális leírását külön-külön értelmezni.
Ennek keretében a „*Híd*" tervezési mintában lévő döntésszerkezet JML alapú formális leírását ismertetem, amelyre egy példát is bemutatok az előző fejezetben megismert egyszerű példaprogramra alapozva. A JML specifikációk helyességét az Iowa State University által fejlesztett JML értelmezővel[5] ellenőriztem.
Ezek alapján megfogalmazom azokat az általános szabályokat, amelyek a JML-es döntésszerkezet szerinti tervezési minta formalizálás alapjai lehetnek.

5.4.1 Híd tervezési minta JML alapú formális leírása

A „*Híd*" tervezési minta JML alapú formális leírását a mintára jellemző döntésszerkezet különböző strukturáltsági szintjeire külön adom meg.
Meg kell különböztetnünk a „*Megvalósítás"* döntés mellékhatásmentes és nem mellékhatásmentes változatait, amely esetekben a formalizáció eltérő.
Nem mellékhatásmentes „*Megvalósítás"* döntés esetén a *műveletMegvalósítás()* művelet (nem */*pure*/* metódusként deklarált) JML alapú viselkedés specifikációjának meg kell jelennie a felhasználási helyen, jelen esetben a *művelet()* metódus JML alapú viselkedés specifikációjában. A metódussal reprezentált döntés által érintett és változtatott adatstruktúrát a JML specifikáció részeként meg kell adni „assignable" deklarációval.
Mellékhatásmentes „*Megvalósítás"* döntés esetén a *műveletMegvalósítás()* művelet (*/*pure*/* metódusként deklarált) JML viselkedésspecifikációja nem kell, hogy megjelenjen a felhasználási helyen, mivel a „pure" metódushívás szerepelhet a felhasználási helyen a JML viselkedés specifikáció részeként, ami a viselkedését a felhasználáskor képviseli. Mellékhatásmentes metódusok esetén látnunk kell, hogy a program állapota nem változhat, amit „*assignable \nothing*" deklarációval kell specifikálnunk.

---

[5] http://sourceforge.net/projects/jmlspecs

A következőkben mellékhatásmentes „Megvalósítás” döntés esetén adom meg a formális leírás lehetőségeit a döntésreprezentáció különböző szintjeire, ahol külön részletezem azon helyzeteket, ahol a nem mellékhatásmenetes esetekben a specifikációban eltérés van.

5.4.1.1 Híd tervezési minta tulajdonságai döntéskiemelés előtt

Ha az *„Elvont ábrázolás”* és a *„Megvalósítás”* döntéslehetőségei nem külön osztályban jelennek meg, továbbá a tartalmazott *„Megvalósítás”* döntés döntéslehetőségei nincsenek külön metódusba szervezve, akkor a következő módon értelmezhető JML-es specifikációjuk:

```
public class ElvontÁbrázolás_Megvalósítás{
/*@ Elvont ábrázolás döntéspredikátum deklaráció@*/
eá_döntéspred;
/*@ Elvont ábrázolás 1. döntéslehetőség adatsruktúra deklarációk@*/
eá_dl1_adatstrukt;
/*@ Elvont ábrázolás 2. döntéslehetőség adatsruktúra deklarációk@*/
eá_dl2_adatstrukt;
/*@ Megvalósítás döntéspredikátum deklaráció@*/
megv_döntéspred;
/*@ Megvalósítás 1. döntéslehetőség adatsruktúra deklarációk@*/
megv_dl1_adatstrukt;
/*@ Megvalósítás 2. döntéslehetőség adatsruktúra deklarációk@*/
megv_dl2_adatstrukt;

/*@ Elvont ábrázolás döntést reprezentáló metódus, amely tartalmazza
a Művelet döntést is. A JML alapú viselkedés specifikációjában
összetett döntésszerkezetet találunk.@*/
/*@public normal_behavior
@requires    eá_döntéspred;
@      {|
@            requires     megv_döntéspred;
@            assignable   eá_dl1_adatstrukt;
@            ensures      eá_dl1_utófelt,
@                         megv_dl1_utófelt;
@      also
@            requires     !megv_döntéspred;
@            assignable   eá_dl1_adatstrukt;
@            ensures      eá_dl1_utófelt,
@                         megv_dl2_utófelt;
@      |}
@also
@requires    !eá_döntéspred;
@      {|
@            requires     megv_döntéspred;
@            assignable   eá_dl2_adatstrukt;
@            ensures      eá_dl2_utófelt,
@                         megv_dl1_utófelt;
@      also
@            requires     !megv_döntéspred;
@            assignable   eá_dl2_adatstrukt;
@            ensures      eá_dl2_utófelt,
@                         megv_dl2_utófelt;
```

```
@       |}
@*/
public void művelet(){
        if(elvont_abrazolas 1. döntéslehetőség dontespredikatum){
                elvont_abrazolas 1. döntéslehetőség műveletek…
                if(művelet megvalósítás 1. döntéslehetőség
                        dontespredikatum){
                        konkrét_megvalósításA döntéslehetőség műveletek…
                }else if(művelet megvalósítás 2. döntéslehetőség
                        dontespredikatum){
                        konkrét_megvalósításB döntéslehetőség műveletek…
                }
        }else if(elvont_abrazolas 2.döntéslehetőségdontespredikatum){
                elvont_abrazolas 2. döntéslehetőség műveletek…
                if(művelet megvalósítás 1. döntéslehetőség
                        dontespredikatum){
                        konkrét_megvalósításA döntéslehetőség műveletek…
                }else if(művelet megvalósítás 2. döntéslehetőség
                        dontespredikatum){
                        konkrét_megvalósításB döntéslehetőség műveletek…
                }
        }
}
```

Látható, hogy mindkét döntés döntéslehetőségeinek adatszerkezetét (*eá_dl1_adatstrukt, eá_dl2_adatstrukt, megv_dl1_adatstrukt, megv_dl2_adatstrukt*) az adott osztályban deklaráljuk. A *művelet()* metódusban kerül megadásra az „*Elvont ábrázolás”* és a „*Megvalósítás”* döntés viselkedése, illetve a metódus JML specifikációjában található az „*Elvont ábrázolás”* döntés viselkedési specifikációja, ami tartalmazza a „*Megvalósítás”* döntés viselkedési követelményeit.

A JML specifikáció annyiban tér el a nem mellékhatásmentes esettől, hogy a „*Megvalósítás”* döntés által érintett döntéslehetőségek JML specifikációjában nem kerülnek megadásra a döntés által érintett *megv_dl1_adatstrukt*, *megv_dl2_adatstrukt* adatstruktúrák, mivel azok a döntés által nem módosultak, tehát a rájuk vonatkozó „*assignable*” deklaráció elmarad.
Nem mellékhatásmentes esetben tehát a művelet() metódus JML specifikációja a következő:

```
/*@    public normal_behavior
@requires    eá_döntéspred;
@      {|
@            requires     megv_döntéspred;
@            assignable   eá_dl1_adatstrukt,
@                         megv_dl1_adatstrukt;
@            ensures      eá_dl1_utófelt,
@                         megv_dl1_utófelt;
@      also
@            requires     !megv_döntéspred;
@            assignable eá_dl1_adatstrukt,
@                         megv_dl2_adatstrukt;
@            ensures      eá_dl1_utófelt,
@                         megv_dl2_utófelt;
@      |}
@also
@requires    !eá_döntéspred;
```

```
@       {|
@               requires      megv_döntéspred;
@               assignable    eá_dl2_adatstrukt,
@                             megv_dl1_adatstrukt;
@               ensures       eá_dl2_utófelt,
@                             megv_dl1_utófelt;
@       also
@               requires      !megv_döntéspred;
@               assignable eá_dl2_adatstrukt,
@                             megv_dl2_adatstrukt;
@               ensures       eá_dl2_utófelt,
@                             megv_dl2_utófelt;
@       |}
@*/
```

Ha az „*Elvont ábrázolás”* és a „*Megvalósítás”* döntések döntéslehetőségei nem külön osztályban jelennek meg, de külön metódusba szervezettek, úgy, hogy a *műveletMegvalósítás()* metódus mellékhatásmentes, akkor a *műveletMegvalósítás()* metódus */*pure*/* metódusként deklarált, ezért a metódushívás megjelenhet a *művelet()* metódus viselkedési specifikációjában úgy, hogy nem szükséges a hivatkozott metódus viselkedési követelményét kifejteni.
Következő módon értelmezhető a JML-es specifikáció:

```
public class ElvontÁbrázolás_Megvalósítás{
/*@ Elvont ábrázolás döntéspredikátum deklaráció@*/
eá_döntéspred;
/*@ Elvont ábrázolás 1. döntéslehetőség adatsruktúra deklarációk@*/
eá_dl1_adatstrukt;
/*@ Elvont ábrázolás 2. döntéslehetőség adatsruktúra deklarációk@*/
eá_dl2_adatstrukt;
/*@ Megvalósítás döntéspredikátum deklaráció@*/
megv_döntéspred;
/*@ Megvalósítás 1. döntéslehetőség adatsruktúra deklarációk@*/
megv_dl1_adatstrukt;
/*@ Megvalósítás 2. döntéslehetőség adatsruktúra deklarációk@*/
megv_dl2_adatstrukt;

/*@ Elvont ábrázolás döntést reprezentáló metódus JML alapú
viselkedés specifikációval. Látható, hogy a műveletMegvalósítás()
mellékhatásmentes metódus szerepelhet a viselkedési
specifikációban.@*/
/*@    public normal_behavior
@               requires      eá_döntéspred;
@               assignable    eá_dl1_adatstrukt;
@               ensures       eá_dl1_utófelt,
@                             műveletMegvalósítás();
@       also
@               requires      !eá_döntéspred;
@               assignable    eá_dl2_adatstrukt;
@               ensures       eá_dl2_utófelt,
@                             műveletMegvalósítás();
@*/
public void művelet(){
        if(elvont_abrazolas 1. döntéslehetőség dontespredikatum){
                elvont_abrazolas 1. döntéslehetőség műveletek…
```

```
        }else if(elvont_abrazolas 2.döntéslehetőség dontespredikatum){
                elvont_abrazolas 2. döntéslehetőség műveletek…
        }
        műveletMegvalósítás();
}

/*@ Művelet döntést reprezentáló mellékhatásmentes – „pure” metódus
JML alapú viselkedés specifikációval.@*/
/*@     public normal_behavior
@               requires      megv_döntéspred;
@               assignable    \nothing;
@               ensures       megv_dl1_utófelt;
@       also
@               requires      !megv _döntéspred;
@               assignable    \nothing;
@               ensures       megv_dl2_utófelt,
@*/
public /*@ pure @*/ void műveletMegvalósítás(){
        if(művelet megvalósítás 1. döntéslehetőség dontespredikatum){
                konkrét_megvalósításA döntéslehetőség műveletek;
        }else if(művelet megvalósítás 2.döntéslehetőség
                dontespredikatum){
                konkrét_megvalósításB döntéslehetőség műveletek;
        }
}

}
```

Mindkét döntés döntéslehetőségeinek adatszerkezetét (*eá_dl1_adatstrukt, eá_dl2_adatstrukt, megv_dl1_adatstrukt, megv_dl2_adatstrukt*) továbbra is az adott osztályban deklaráljuk. A *művelet()* metódusban kerül megadásra az „*Elvont ábrázolás”* döntés. A „*Megvalósítás”* döntés külön metódusban (*műveletMegvalósítás()*) deklarált.

Ha a tartalmazott „*Megvalósítás”* döntést deklaráló *műveletMegvalósítás()* metódus nem mellékhatásmentes, akkor *műveletMegvalósítás()* metódus viselkedés specifikációjának ugyanúgy meg kell jelennie az „*Elvont ábrázolás”* döntést reprezentáló *művelet()* metódus viselkedés specifikációjában, mintha a két metódus megvalósítása az előző szinttel egyező módon együtt lett volna definiálva.
A *műveletMegvalósítás()* JML specifikációja nem mellékhatásmentes esetben annyival tér el a mellékhatásmentes esettől, hogy az állapotátmenetben érintett adatstruktúra megadásra kerül, illetve nem „pure” metódusként kell deklarálni:

```
/*@ Művelet döntést reprezentáló nem mellékhatásmentes metódus JML
alapú viselkedés specifikációval.@*/
/*@     public normal_behavior
@       requires      megv_döntéspred;
@       assignable    megv_dl1_adatstrukt;
@       ensures       megv_dl1_utófelt;
@also
@       requires      !megv _döntéspred;
@       assignable    megv_dl2_adatstrukt;
@       ensures       megv_dl2_utófelt,
@*/
```

```
private void műveletmegvalósítás(){…}
```

Ha az „*Elvont ábrázolás*” és a „*Megvalósítás*” döntések megvalósításait külön osztályba szervezve deklaráljuk és a „*Művelet*” döntést deklaráló *műveletMegvalósítás()* metódus mellékhatásmentes, akkor a *műveletMegvalósítás()* metódus viselkedés specifikációjának nem kell a felhasználási helyen (*művelet()* viselkedési specifikációjában) megjelennie, elég a metódushívásra hivatkozni:

```
public class ElvontÁbrázolás {
/*@ Elvont ábrázolás döntéspredikátum deklaráció@*/
eá_döntéspred;
/*@ Elvont ábrázolás 1. döntéslehetőség adatsruktúra deklarációk@*/
eá_dl1_adatstrukt;
/*@ Elvont ábrázolás 2. döntéslehetőség adatsruktúra deklarációk@*/
eá_dl2_adatstrukt;

private Megvalósítás      megvalósítás;

public void setMegvalósítás(Megvalositas megvalositas){
      this.megvalositas=megvalositas;
}

/*@private normal_behavior
@     assignable   \nothing;
@     ensures      \result==megvalósítás;
@*/
public /*@ pure @*/ Megvalósítás getMegvalósítás(){
      return megvalositas;
}

/*@ Elvont ábrázolás döntést reprezentáló metódus JML alapú
viselkedés specifikációval.
Látható, hogy a műveletMegvalósítás() mellékhatásmentes metódus
szerepelhet a viselkedési specifikációban.@*/
/*@public normal_behavior
@     requires     eá_döntéspred;
@     assignable   eá_dl1_adatstrukt;
@     ensures      eá_dl1_utófelt,
@                  getMegvalósítás().műveletMegvalósítás();
@also
@     requires     !eá_döntéspred;
@     assignable   eá_dl2_adatstrukt;
@     ensures      eá_dl2_utófelt,
@                  getMegvalósítás().műveletMegvalósítás();
@*/
public void művelet(){
      if(elvont_abrazolas 1. döntéslehetőség dontespredikatum){
            elvont_abrazolas 1. döntéslehetőség műveletek;
      }else if(elvont_abrazolas 2.döntéslehetőség dontespredikatum){
            elvont_abrazolas 2. döntéslehetőség műveletek;
      }
      getMegvalósítás().műveletMegvalósítás();
}

}
```

```
public class Megvalósítás {
/*@ Megvalósítás döntéspredikátum deklaráció@*/
megv_döntéspred;
/*@ Megvalósítás 1. döntéslehetőség adatsruktúra deklarációk@*/
megv_dl1_adatstrukt;
/*@ Megvalósítás 2. döntéslehetőség adatsruktúradeklarációk@*/
megv_dl2_adatstrukt;

/*@ Művelet döntést reprezentáló mellékhatásmentes – „pure” metódus
JML alapú viselkedés specifikációval.@*/
/*@public normal_behavior
@      requires     megv_döntéspred;
@      assignable   \nothing;
@      ensures      megv_dl1_utófelt;
@also
@      requires     !megv _döntéspred;
@      assignable   \nothing;
@      ensures      megv_dl2_utófelt,
@*/
public void /*@ pure @*/ műveletMegvalósítás(){
       if(művelet megvalósítás 1. döntéslehetőség dontespredikatum){
              KonkrétMegvalósításA döntéslehetőség műveletek;
       }else if(művelet megvalósítás 2. döntéslehetőség
                 dontespredikatum){
              KonkrétMegvalósításB döntéslehetőség műveletek;
       }
}

}
```

Látható, hogy a két döntés döntéslehetőségeinek adatszerkezetének deklarálására a vonatkozó döntéseknek megfelelő *ElvontÁbrázolás* (*eá_dl1_adatstrukt, eá_dl2_adatstrukt*) és a *Megvalósítás* (*megv_dl1_adatstrukt, megv_dl2_adatstrukt*) osztályokban kerül sor.
Az *„Elvont ábrázolás”* döntés viselkedését az *ElvontÁbrázolás.művelet()* metódusban deklaráljuk. A metódus JML alapú viselkedési specifikációjában az *„Elvont ábrázolás”* döntés viselkedését specifikáljuk, amely tartalmazza a mellékhatásmentes *műveletMegvalósítás()* metódushívást, hivatkozva ezzel annak viselkedési követelményeire.
A *„Megvalósítás”* döntés viselkedését a *Megvalósítás.műveletMegvalósítás() „pure”* metódusban deklaráljuk.

Ha a *„Megvalósítás”* döntést deklaráló *műveletMegvalósítás()* metódus nem mellékhatásmentes, akkor a *műveletMegvalósítás()* metódus viselkedés specifikációjának továbbra is meg kell jelennie a felhasználási helyen a *művelet()* viselkedési specifikációjában.

```
/*@ public normal_behavior
@requires eá_döntéspred;
@      {|
@             requires     getMegvalósítás().megv_döntéspred;
@             assignable   eá_dl1_adatstrukt,
@                          getMegvalósítás().megv_dl1_adatstrukt;
```

```
@            ensures      eá_dl1_utófelt,
@                         getMegvalósítás().megv_dl1_utófelt;
@      also
@            requires     !getMegvalósítás().megv_döntéspred;
@            assignable   eá_dl1_adatstrukt,
@                         getMegvalósítás().megv_dl2_adatstrukt;
@            ensures      eá_dl1_utófelt,
@                         getMegvalósítás().megv_dl2_utófelt;
@      |}
@also
@requires !eá_döntéspred;
@      {|
@            requires     getMegvalósítás().megv_döntéspred;
@            assignable   eá_dl2_adatstrukt,
@                         getMegvalósítás().megv_dl1_adatstrukt;
@            ensures      eá_dl2_utófelt,
@                         getMegvalósítás().megv_dl1_utófelt;
@      also
@            requires     !getMegvalósítás().megv_döntéspred;
@            assignable   eá_dl2_adatstrukt,
@                         getMegvalósítás().megv_dl2_adatstrukt;
@            ensures      eá_dl2_utófelt,
@                         getMegvalósítás().megv_dl2_utófelt;
@      |}
@*/
public void művelet(){…}
```

Nem mellékhatásmentes esetben a *Megvalósítás.műveletMegvalósítás()* metódus JML specifikációja az előző szinthez képest nem változik.

5.4.1.2 Híd tervezési minta tulajdonságai döntéskiemelés után

Az érintett döntések kiemelt állapotában az „*Elvont ábrázolás”* és a „*Megvalósítás”* döntéseket osztályhierarchiába szervezve deklaráljuk.

A „*Megvalósítás”* döntést reprezentáló *műveletMegvalósítás()* metódus mellékhatásmentes JML alapú viselkedés specifikációja döntés kiemelés után is eltér a nem mellékhatásmentes viselkedés specifikációtól.

A „*Megvalósítás”* döntéslehetőségeit a „*Megvalósítás”* osztály alosztályai reprezentálják, a nem döntés specifikus részeket „*Megvalósítás”* szülőosztály adja meg.

Az „*Elvont ábrázolás”* döntést szintén osztályhierarchiába deklaráltan adjuk meg, ahol a döntéslehetőségeket az egyes alosztályok reprezentálják. A nem döntés specifikus részeket az „*ElvontÁbrázolás”* szülőosztályban adjuk meg.

Ha a „*Megvalósítás”* döntés osztályhierarchiában a döntést deklaráló *műveletMegvalósítás()* polimorf metódus mellékhatásmentes, akkor a metódus viselkedését nem szükséges a hívási oldalon specifikálni, csupán a metódushívást kell a viselkedés specifikációban megjeleníteni.

```
public abstract class ElvontÁbrázolás {
/*@ Elvont ábrázolás döntéspredikátum és döntéslehetőségek modell
adatsruktúra deklarációi@*/
/*@
```

```
@ public model instance eá_döntéspred_modell;
@ public model instance eá_dl1_adatstrukt_modell;
@ public model instance eá_dl2_adatstrukt_modell;
@*/

private Megvalósítás megvalósítás;

public void setMegvalósítás(Megvalositas megvalositas){
	this.megvalositas=megvalositas;
}

/*@private normal_behavior
@assignable \nothing;
@ensures \result==megvalósítás;
@*/
public /*@ pure @*/ Megvalósítás getMegvalósítás(){
	return megvalositas;
}

/*@ Elvont ábrázolás döntést reprezentáló metódus JML alapú
viselkedés specifikációval.
A döntéspredikátumok, döntéslehetőség specifikus adatstruktúrák
modell változókkal reprezentáltak. Az utófeltételek a modell
változókra építve deklaráltak.
A műveletMegvalósítás()mellékhatásmentes metódus, ezért szerepelhet
a viselkedés specifikációban is.
@*/
/*@
@public normal_behavior
@	requires	eá_döntéspred_modell;
@	assignable	eá_dl1_adatstrukt_modell;
@	ensures		eá_dl1_utófelt_modell,
@			getMegvalósítás().műveletMegvalósítás();
@also
@public normal_behavior
@	requires	!eá_döntéspred_modell;
@	assignable	eá_dl2_adatstrukt_modell;
@	ensures		eá_dl2_utófelt_modell,
@			getMegvalósítás().műveletMegvalósítás();
@*/
public abstract void művelet();

}
```

Látható, hogy az „*ElvontÁbrázolás*” döntéslehetőség specifikus adatstruktúrák és a döntéspredikátum modell változóként jelenik meg a döntés struktúra szülőosztályában.
(*eá_döntéspred_modell , eá_dl1_adatstrukt_modell, eá_dl2_adatstrukt_modell*)
Ezeket a modell változókat használjuk fel arra, hogy a döntéslehetőségek szerinti viselkedést a döntést reprezentáló *művelet()* absztrakt, polimorf metódus viselkedési specifikációjában megadjuk.
A „*Megvalósítás*” döntés viselkedését – mivel művelete mellékhatásmentes – a *megvalósítás.műveletMegvalósítás()* metódushívással adhatjuk meg a *művelet()* metódus viselkedés specifikációjában.

Ha a *„Megvalósítás”* döntés osztályhierarchiában a döntést deklaráló *műveletMegvalósítás()* polimorf metódus nem mellékhatásmentes, akkor a metódus viselkedését a hívási oldalon is specifikálni kell. Ezekben az esetekben a hívási oldal JML alapú viselkedés specifikációjában többszintű döntések (*„Megvalósítás”* döntés beágyazva az *„Elvont ábrázolás”* döntésbe) jelennek meg:

```
/*@ public normal_behavior
@requires    eá_döntéspred_modell;
@      {|
@            requires     megvalósítás.megv_döntéspred_modell;
@            assignable   eá_dl1_adatstrukt_modell,
@                  getMegvalósítás().megv_dl1_adatstrukt_modell;
@            ensures      eá_dl1_utófelt_modell,
@                  getMegvalósítás().megv_dl1_utófelt_modell;
@      also
@            requires     !megvalósítás.megv_döntéspred_modell;
@            assignable   eá_dl1_adatstrukt_modell,
@                  getMegvalósítás().megv_dl2_adatstrukt_modell;
@            ensures eá_dl1_utófelt_modell,
@                  getMegvalósítás().megv_dl2_utófelt_modell;
@      |}
@also
@requires !eá_döntéspred_modell;
@      {|
@            requires     getMegvalósítás().megv_döntéspred_modell;
@            assignable   eá_dl2_adatstrukt_modell,
@                  getMegvalósítás().megv_dl1_adatstrukt_modell;
@            ensures      eá_dl2_utófelt_modell,
@                  getMegvalósítás().megv_dl1_utófelt_modell;
@      also
@            requires     !megvalósítás.megv_döntéspred_modell;
@            assignable   eá_dl2_adatstrukt_modell,
@                  getMegvalósítás().megv_dl2_adatstrukt_modell;
@            ensures      eá_dl2_utófelt_modell,
@                  getMegvalósítás().megv_dl2_utófelt_modell;
@      |}
@*/
public abstract void művelet();
```

A *művelet()* metódusban megadott viselkedés a következő alosztályokban „hasad” ketté az *Elvont ábrázolás* döntés szerint.

```
public class FinomitottElvontÁbrázolás1 extends ElvontÁbrázolás{
/*@ Elvont ábrázolás 1. döntéslehetőség adatsruktúra deklarációk@*/
eá_dl1_adatstrukt;

/*@
@ public instance represents eá_döntéspred_modell <- true;
@ public instance represents
@                  eá_dl1_adatstrukt_modell <-eá_dl1_adatstrukt;
@*/

/*@
@ also
@ public normal_behavior
```

```
@       assignable    eá_dl1_adatstrukt;
@       ensures       eá_dl1_utófelt,
@                     getMegvalósítás().műveletMegvalósítás();
@*/
public void művelet(){
        elvont_abrazolas 1. döntéslehetőség műveletek;
        getMegvalósítás().műveletMegvalósítás();
        }
}

public class FinomitottElvontÁbrázolás2 extends ElvontÁbrázolás{
/*@ Elvont ábrázolás 2. döntéslehetőség adatsruktúra deklarációk@*/
eá_dl2_adatstrukt;

/*@
@ public instance represents eá_döntéspred_modell <- false;
@ public instance represents
@                     eá_dl2_adatstrukt_modell <- eá_dl2_adatstrukt;
@*/

/*@
@ also
@ public normal_behavior
@       assignable    eá_dl2_adatstrukt;
@       ensures       eá_dl2_utófelt,
@                     getMegvalósítás().műveletMegvalósítás();
@*/
public void művelet(){
        elvont_abrazolas 2. döntéslehetőség műveletek;
        getMegvalósítás().műveletMegvalósítás();
        }
}
```

Az „*ElvontÁbrázolás”* döntéslehetőség specifikus adatstruktúrája a szülőosztályban modell változókként jelent meg. A *FinomítottElvontÁbrázolás1, FinomítottElvontÁbrázolás2* alosztályokban konkrét implementációs változókkal reprezentáltak. Hogy melyik modell változót melyik tag változó valósítja meg, a *represents* JML specifikációs állítással adható meg.
Az „*ElvontÁbrázolás”* döntés döntéslehetőség specifikus viselkedését a két alosztály *művelet()* metódus JML specifikációjában lehet megadni, ahol látható, hogyan válik ketté a döntéslehetőségek-alosztályok szerinti viselkedése. A *művelet()* JML viselkedés specifikációjában a „*Megvalósítás”* döntést reprezentáló mellékhatásmentes *műveletMegvalósítás()* metódushívás szerepelhet.

A nem mellékhatásmentes *műveletMegvalósítás()* metódus esetén a JML specifikációja a *művelet()* metódusnak a következők szerint változik:

```
public class FinomitottElvontÁbrázolás1 extends ElvontAbrazolas{
…
/*@
@also
@public normal_behavior
@       requires      megvalósítás.megv_döntéspred_modell;
@       assignable    eá_dl1_adatstrukt,
```

```
@                   getMegvalósítás().megv_dl1_adatstrukt_modell;
@      ensures      eá_dl1_utófelt,
@                   getMegvalósítás().megv_dl1_utófelt_modell;
@also
@      requires     !megvalósítás.megv_döntéspred_modell;
@      assignable   eá_dl1_adatstrukt,
@                   getMegvalósítás().megv_dl2_adatstrukt_modell;
@      ensures      eá_dl1_utófelt,
@                   getMegvalósítás().megv_dl2_utófelt_modell;
@*/
public void művelet(){…)
}

public class FinomitottElvontÁbrázolás2 extends ElvontAbrazolas{
…
/*@
@also
@public normal_behavior
@      requires     megvalósítás.megv_döntéspred_modell;
@      assignable   eá_dl2_adatstrukt,
@                   getMegvalósítás().megv_dl1_adatstrukt_modell;
@      ensures      eá_dl2_utófelt,
@                   getMegvalósítás().megv_dl1_utófelt_modell;
@also
@      requires     !megvalósítás.megv_döntéspred_modell;
@      assignable eá_dl2_adatstrukt,
@                   getMegvalósítás().megv_dl2_adatstrukt_modell;
@      ensures      eá_dl2_utófelt,
@                   getMegvalósítás().megv_dl2_utófelt_modell;
@*/
public void művelet(){…}
}
```

A *művelet()* JML viselkedés specifikációjában megjelennek a *„Megvalósítás”* döntés döntéslehetőségei, melyek által érintett adatstruktúra modell változókkal reprezentált.

A *„Megvalósítás”* döntéskiemelés után a *Megvalósítás* szülő osztály és annak *KonkrétMegvalósításA* és *KonkrétMegvalósításB* alosztályaiban jelenik meg.

```
public abstract class Megvalósítás {

/*@ Megvalósítás döntéspredikátum és döntéslehetőségeinek modell
adatsruktúra deklarációi@*/
/*@
@ public model instance megv_döntéspred_modell;
@ public model instance megv_dl1_adatstrukt_modell;
@ public model instance megv_dl2_adatstrukt_modell;
@*/

/*@ Megvalósítás döntést reprezentáló metódus JML alapú viselkedés
specifikációval.
A döntéspredikátumok, döntéslehetőség specifikus adatstruktúrák
modell változókkal reprezentáltak. Az utófeltételek a modell
változókra építve deklaráltak.@*/
/*@
@public normal_behavior
```

```
@      requires     megv_döntéspred_modell;
@      assignable   \nothing;
@      ensures      megv_dl1_utófelt_modell,
@also
@public normal_behavior
@      requires     !megv_döntéspred_modell;
@      assignable   \nothing;
@      ensures      megv_dl2_utófelt_modell,
@*/
public abstract void /*@ pure @*/ műveletMegvalósítás();
}
```

Az „*Megvalósítás”* döntéslehetőség specifikus adatstruktúrák és a döntéspredikátum modell változóként jelennek meg a döntésstruktúra szülő osztályában. (*megv_döntéspred_modell, megv_dl1_adatstrukt_modell, megv_dl2_adatstrukt_modell*)
Ezeket a modell változókat használjuk fel arra, hogy a döntéslehetőségek szerinti viselkedést a döntést reprezentáló *műveletMegvalósítás()* absztrakt, polimorf metódus viselkedési specifikációjában megadjuk.
Mellékhatásmentes *műveletMegvalósítás()* metódus esetén az *assignable*-lel nem deklarálunk olyan adatstruktúrát, amely értéke változna.

Nem mellékhatásmentes *műveletMegvalósítás()* metódus esetén *assignable*-lel deklaráljuk azokat a változókat, amelyeket érint az állapotváltozás:

```
public abstract class Megvalósítás {
…
/*@
@public normal_behavior
@      requires     megv_döntéspred_modell;
@      assignable   megv_dl1_adatstrukt_modell;
@      ensures      megv_dl1_utófelt_modell,
@also
@public normal_behavior
@      requires     !megv_döntéspred_modell;
@      assignable   megv_dl2_adatstrukt_modell;
@      ensures      megv_dl2_utófelt_modell,
@*/
public abstract void műveletMegvalósítás();
}
```

A *műveletMegvalósítás()* metódusban megadott viselkedés a következő alosztályokban „hasad” ketté a „*Megvalósítás”* döntés szerint:

```
public class KonkretMegvalositasA extends Megvalósítás{
/*@ Megvalósítás 1. döntéslehetőség adatsruktúra deklarációk@*/
megv_dl1_adatstrukt;

/* A döntéspredikátum a döntéslehetőségnek megfelelő alosztály
példányosítása során konkrét értékkel reprezentált.
Döntéslehetőségspecifikus változók valós reprezentációhoz kötése.*/
/*@
@ public instance represents megv_döntéspred_modell <- true;
```

```
@ public instance represents
@            megv_dl1_adatstrukt_modell <- megv_dl1_adatstrukt;
@*/

/* Döntéslehetőség specifikus műveletMegvalósítás() implementáció, a
hozzá tartozó JML alapú viselkedés specifikációval.*/
/*@
@also
@      public normal_behavior
@            assignable   \nothing;
@            ensures      megv_dl1_utófelt_modell,
@*/
public void műveletMegvalósítás(){
       KonkrétMegvalósításA döntéslehetőség műveletek…
       }
}

public class KonkrétMegvalósításB extends Megvalósítás{
/*@ Megvalósítás 2. döntéslehetőség adatsruktúra deklarációk@*/
megv_dl2_adatstrukt;

/* A döntéspredikátum a döntéslehetőségnek megfelelő alosztály
példányosítása során konkrét értékkel reprezentált.
Döntéslehetőségspecifikus változók valós reprezentációhoz kötése.*/
/*@
@ public instance represents megv_döntéspred_modell <- false;
@ public instance represents
@            megv_dl2_adatstrukt_modell <- megv_dl2_adatstrukt;
@*/

/* Döntéslehetőség specifikus műveletMegvalósítás() implementáció, a
hozzá tartozó JML alapú viselkedés specifikációval.*/
/*@
@also
@public normal_behavior
@      assignable   \nothing;
@      ensures      megv_dl2_utófelt_modell,
@*/
public void műveletMegvalósítás(){
       KonkrétMegvalósításB döntéslehetőség műveletek…
       }
}
```

Az *„Megvalósítás”* döntéslehetőség specifikus adatstruktúrája a szülőosztályban modell változóként jelent meg. A *KonkrétMegvalósításA, KonkrétMegvalósításB* alosztályokban implementációs változókkal reprezentáltak. Hogy melyik modell változót melyik tag változó valósítja meg, a *represents* JML specifikációs állítással adható meg.

Látnunk kell, hogy a *„Megvalósítás”* döntés döntéslehetőség specifikus viselkedését a két alosztály *műveletMegvalósítás()* metódus JML specifikációjában adjuk meg, ahol látható, hogyan válik ketté a döntéslehetőségek-alosztályok szerinti viselkedése. Mellékhatásmentes – *„pure”* metódus esetén az adatstruktúrák állapota nem változhat, amit az *assignable \nothing* kifejezéssel specifikálunk.

Nem mellékhatásmentes *műveletMegvalósítás()* metódus esetén a JML specifikációk a következők szerint változnak:

```
public class KonkretMegvalositasA extends Megvalósítás{
…
/*@
@also
@public normal_behavior
@      assignable megv_dl1_adatstrukt_modell;
@      ensures megv_dl1_utófelt_modell,
@*/
public void műveletMegvalósítás(){…}
}

public class KonkrétMegvalósításB extends Megvalósítás{
       …
       /*@
       @also
       @public normal_behavior
       @      assignable   megv_dl2_adatstrukt_modell;
       @      ensures      megv_dl2_utófelt_modell,
       @*/
       public void műveletMegvalósítás(){}
}
```

5.4.2 Híd tervezési minta JML alapú formális leírása példán keresztül

A következőkben a „*Híd*” tervezési minta JML alapú formális leírására látunk példát. A példa a tervezési mintában lévő döntésszerkezet-döntések két állapotát mutatja be, amelyből láthatóvá válik, hogy a „*Híd*” tervezési mintánál megadott JML alapú formalizációs követelmények miként érvényesülnek.
A szemléltetéshez már korábban felhasznált egyszerű vásárlást-fizetést megvalósító példaprogram döntésszerkezet szerinti két állapotát egészítettem ki JML alapú viselkedés specifikációval.

5.4.2.1 Híd tervezési minta tulajdonságai döntéskiemelés előtt

A „*Vásárlás”* és „*Fizetés”* döntések megvalósításai külön osztályokban (*Vasarlas*, *Fizetes*) jelennek meg, melyek szerint látni fogjuk azokat az ekvivalens döntéspredikátumú döntéseket, melyek összevonásával és kiemelésével az osztályhierarchiába szervezés indokolt lesz.

```
package hu.dontes.szint1;
import hu.dontes.szint1.Fizetes;
//@ model import org.jmlspecs.models.*;

/** Vasarlasi es fizetesi adatok bekerese, vasarlas es fizetes
 *  modjanak megfelelo bizonylat nyomtatasa.
 */
public class Vasarlas {
/**    Modell fuggvennyel vizsgaljuk az bejovo argumentumokat: */
/*@    public static pure model boolean parseable( String s ) {
@      try { int d = Integer.parseInt(s); return true; }
```

```
@       catch (Exception e) { return false; }}
@*/

/**    Vasarlas dontespredikatuma: */
private /*@ spec_public @*/ boolean szemelyesVasarlas=true;

/**    Fizetesi feltetelek kezeleset objektum osszetetellel
 * mas komponenshez delegaljuk.*/
private /*@ spec_public @*/ Fizetes fizetes;
//@ public instance invariant fizetes != null;

/**    Szemelyes vasarlas eseten szukseges adatok:   */
private /*@ spec_public @*/ String keresztnev = "";
private /*@ spec_public @*/ String vezeteknev = "";

/**    Ceges vasarlas eseten szukseges adatok:   */
private /*@ spec_public @*/ String cegNev = "";
private /*@ spec_public @*/ String cegSzekhely = "";

/*@
@public normal_behavior
@       assignable \nothing;
@       ensures \result instanceof Vasarlas;
@*/
public static Vasarlas getVasarlas(String[] args){
        return new Vasarlas();
}

public static void main(String[] args) {
        //Ha az argumentum szam nem megfelelo.
        if(args.length < 7 ){
                System.err.println("Az argumentum szam nem " +
                                "megfelelo!");
                System.exit(-1);
        }
        try{
                //Vasarlast kezelo objektum letrehozasa.
                Vasarlas vasarlas=Vasarlas.getVasarlas(args);
                //Vasarlasi adatok megadasa,
                vasarlas.setVasarlasInfo(args);
                //Fizetesi adatok megadasa.
                vasarlas.setFizetes(args);
                //Bizonylat nyomtatasa.
                System.out.println(vasarlas.getBizonylat(args));

                }
        catch (java.lang.NumberFormatException nfe){
                System.err.println("Az argumentumok formatuma nem "+
                                        "megfelelo!");
                System.exit(-1);
        }
}

/**    Vasarlasi adatok vasarlas tipusanak megfelelo beallitasa.
 */
/*@
@public normal_behavior
```

```
@      requires args[0]!="nem";
@      assignable szemelyesVasarlas,vezeteknev,keresztnev;
@      ensures szemelyesVasarlas==true && vezeteknev==args[1]&&
@             keresztnev==args[2];
@also
@public normal_behavior
@      requires args[0]=="nem";
@      assignable szemelyesVasarlas, cegNev, cegSzekhely;
@      ensures szemelyesVasarlas==false&&cegNev==args[1]&&
@             cegSzekhely == args[2];
@*/
public void setVasarlasInfo(String[] args){
       if(!args[0].equals("nem")){
              szemelyesVasarlas=true;
              System.out.println("Szemelyes vasarlass?: igen");
              vezeteknev = args[1];
              System.out.println("Vezeteknev: " + vezeteknev);
              keresztnev = args[2];
              System.out.println("Keresztnev: " + keresztnev);
       } else {
              szemelyesVasarlas=false;
              System.out.println("Szemelyes vasarlass?: nem");
              cegNev = args[1];
              System.out.println("Ceg neve: " + cegNev);
              cegSzekhely = args[2];
              System.out.println("Ceg szekhelye: " + cegSzekhely);
       }
}

/**    Fizetes modjanak meghatarozasa, a fizetes modjanak
 *  megfelelo adatok bekerese.
 */
/*@
@public normal_behavior
@      requires args[3]!="nem" && parseable(args[4])&&
@             parseable(args[5]) && parseable(args[6]);
@      assignable fizetes,fizetes.keszpenzesFizetes,
@             fizetes.bizonylatAzonosito, fizetes.fizetendoOsszeg,
@             fizetes.visszajaro, System.out;
@      ensures fizetes instanceof Fizetes &&
@             fizetes.keszpenzesFizetes==true &&
@             fizetes.bizonylatAzonosito==Integer.parseInt(args[2])&&
@             fizetes.fizetendoOsszeg== Integer.parseInt(args[5]) &&
@             fizetes.visszajaro == Integer.parseInt(args[6]);
@also
@public normal_behavior
@      requires args[3]=="nem" &&parseable(args[2])&&
@              parseable(args[5]) && parseable(args[6]);
@      assignable fizetes.keszpenzesFizetes,
@             fizetes.bizonylatAzonosito,fizetes.fizetendoOsszeg,
@             fizetes.kartyaszam, System.out;
@      ensures fizetes instanceof Fizetes &&
@             fizetes.keszpenzesFizetes == false &&
@             fizetes.bizonylatAzonosito==Integer.parseInt(args[2]) &&
@             fizetes.fizetendoOsszeg== Integer.parseInt(args[5]) &&
@             fizetes.kartyaszam==args[6];
@also
```

```
@public exceptional_behavior
@      requires !parseable(args[2])|||!parseable(args[5])||
@              !parseable(args[6]);
@      assignable \nothing;
@      signals_only java.lang.NumberFormatException;
@*/
public void setFizetes(String[] args)
                              throws java.lang.NumberFormatException{
       fizetes = new Fizetes();
       fizetes.setFizetesInfo(args);
}

/** Fizetes modjanak megfelelo referencia elkerese.
 */
/*@public normal_behavior
@      assignable \nothing;
@      ensures \result == fizetes;
@*/
public /*@ pure @*/ Fizetes getFizetes(){
       return fizetes;
}

/** A vasarlas es fizetes adatainak listazasa.
 */
/*@
@public normal_behavior
@      requires szemelyesVasarlas;
@      assignable \nothing;
@      ensures \result == "Vezeteknev: " + vezeteknev +
@             "; Keresztnev: " + keresztnev +
@             "\nFizetesi adatok: "+getFizetes().getFizetesInfo(args);
@also
@public normal_behavior
@      requires !szemelyesVasarlas;
@      assignable \nothing;
@      ensures \result == "Ceg neve: " + cegNev +
@      "; Ceg cime: " + cegSzekhely+
@      "\nFizetesi adatok: "+ getFizetes().getFizetesInfo(args);
@*/
public String getBizonylat(String[] args){
       if(szemelyesVasarlas)
              return "Bizonylat: \n"+ "Vezeteknev: "+vezeteknev+
              "; Keresztnev: "+keresztnev+
              "\nFizetesi adatok: "+getFizetes().getFizetesInfo(args);
       else
              return "Bizonylat: \n"+"Ceg neve: "+cegNev+"; " +
              "Ceg szekhelye: " + cegSzekhely+
              "\nFizetesi adatok: "+getFizetes().getFizetesInfo(args);
}

}
```

A „*Vásárlás*” döntéssel kapcsolatosan két ekvivalens döntéspredikátumú döntést találunk, melyek összevonása indokolt lehet:

```
/**    Vasarlasi adatok vasarlas tipusanak megfelelo beallitasa.
 */
```

```
/*@
@public normal_behavior
@      requires args[0]!="nem";
@      assignable szemelyesVasarlas,vezeteknev,keresztnev;
@      ensures szemelyesVasarlas==true && vezeteknev==args[1]&&
@             keresztnev==args[2];
@also
@public normal_behavior
@      requires args[0]=="nem";
@      assignable szemelyesVasarlas, cegNev, cegSzekhely;
@      ensures szemelyesVasarlas==false && cegNev==args[1]&&
@             cegSzekhely == args[2];
@*/
public void setVasarlasInfo(String[] args){…}
```

Vásárlás típusának megfelelően állítjuk be a vásárlással kapcsolatos adatokat, mivel személyes és céges vásárlás esetén más adatokat kell bekérni. A döntéspredikátum:

```
      args[0]!="nem",args[0]=="nem"
```

Ezzel a másik döntés döntéspredikátuma ekvivalens, mivel döntéspredikátumának kifejezésében a *szemelyesVasarlas* logikai változó szerepel, amit az első vásárlással kapcsolatos döntés döntéslehetőségeiben állítunk be:

```
/*@
@public normal_behavior
@      requires szemelyesVasarlas;
@      assignable \nothing;
@      ensures \result == "Vezeteknev: " + vezeteknev +
@      "; Keresztnev: " + keresztnev +
@      "\nFizetesi adatok: "+getFizetes().getFizetesInfo(args);
@also
@public normal_behavior
@      requires !szemelyesVasarlas;
@      assignable \nothing;
@      ensures \result == "Ceg neve: " + cegNev +
@      "; Ceg cime: " + cegSzekhely+
@      "\nFizetesi adatok: "+getFizetes().getFizetesInfo(args);
@*/
public String getBizonylat(String[] args){…}
```

Mivel a két vásárlással kapcsolatos döntés döntéspredikátuma ekvivalens, ezért az összevonásuk-kiemelésük indokolt. („3.2 A döntésredundanciák elkerülésének esetei”, „4.4 Döntésösszevonás eseteinek vizsgálata JML-lel”)
Látnunk kell, hogy a döntés nem döntés specifikus része tartalmaz egy másik döntést („*Fizetés”*), amely döntéslehetőségeinek megfelelően egészül ki a „*Vásárlás”* döntés *getBizonylat()* műveletének viselkedése.
A tartalmazott döntés döntéslehetőségei mellékhatásmentesek, ezért nem jelennek meg a tartalmazó döntésben, azokat a mellékhatásmentes metódushívással szemléltetjük a JML specifikációban.

A tartalmazott „*Fizetés”* döntés megvalósítása JML alapú viselkedés specifikációval a következő:

```
package hu.dontes.szint1;
//@ model import org.jmlspecs.models.*;
```

```
/** Fizetes modja lehet keszpenz vagy bankkartya alapu, amely szerint
 * mas adatok bekerese, nyomtatasa szukseges.
 */
public class Fizetes{
/**    Modell fuggvennyel vizsgaljuk az bejovo argumentumokat: */
/*@    public static pure model boolean parseable( String s ) {
@      try { int d = Integer.parseInt(s); return true; }
@      catch (Exception e) { return false; }
@      }
@*/
/**    Fizetes dontespredikatuma: */
private /*@ spec_public @*/ boolean keszpenzesFizetes = true;
//@ public instance initially keszpenzesFizetes == true;

/**    Keszpenzes es bankkartyas fizetes eseten is szukseges
 *  megadni:*/
private /*@ spec_public @*/ long bizonylatAzonosito = 0;
//@ public instance initially bizonylatAzonosito == 0;
private /*@ spec_public @*/ int fizetendoOsszeg=0;
//@ public instance initially fizetendoOsszeg == 0;

/**    Keszpenzes fizetes eseten szukseges megadni: */
private /*@ spec_public @*/ int visszajaro = 0;
//@ public instance initially visszajaro == 0;

/**    Bankkartyas fizetes eseten szukseges megadni: */
private /*@ spec_public @*/ String kartyaszam = "";
//@ public instance initially kartyaszam == "";

/**
 *  Megkapja a fizetendo osszeget es a bizonylat szamot,
 *  amelyek a fizetes modjatol fuggetlen adatok.
 *  A visszajaro es a kartyaszam a fizetes modjatol fuggo adatok.
 */
/*@
@public normal_behavior
@      requires args[3]!="nem" &&
@             parseable(args[4])&&parseable(args[5])&&
@             parseable(args[6]);
@      assignable keszpenzesFizetes, bizonylatAzonosito,
@             fizetendoOsszeg, visszajaro, System.out;
@      ensures keszpenzesFizetes == true &&
@             bizonylatAzonosito == Integer.parseInt(args[4])&&
@             fizetendoOsszeg == Integer.parseInt(args[5])&&
@             visszajaro == Integer.parseInt(args[6]);
@also
@public normal_behavior
@      requires args[3]=="nem" && parseable(args[4])&&
@             parseable(args[5]) && parseable(args[6]);
@      assignable keszpenzesFizetes, bizonylatAzonosito,
@             fizetendoOsszeg,kartyaszam, System.out;
@      ensures keszpenzesFizetes == false &&
@             bizonylatAzonosito == Integer.parseInt(args[4]) &&
@             fizetendoOsszeg ==Integer.parseInt(args[5])&&
@             kartyaszam == args[6];
@also
```

```
@public exceptional_behavior
@      requires !parseable(args[4])|||!parseable(args[5])||
@             !parseable(args[6]);
@      assignable \nothing;
@      signals_only java.lang.NumberFormatException;
@*/
public void setFizetesInfo(String[] args)
                                    throws NumberFormatException{
       System.out.println("Keszpenzes fizetes?:(igen/nem) "+args[3]);
       /*Keszpenzes es bankkartyas fizetes eseten is szukseges
       adatok*/
       bizonylatAzonosito = Integer.parseInt(args[4]);
       System.out.println("Bizonylat azonosito:(szam) "
                                                + bizonylatAzonosito);
       fizetendoOsszeg = Integer.parseInt(args[5]);
       System.out.println("Fizetendo osszeg:(szam) "
                                                + fizetendoOsszeg);
       if(!args[3].equals("nem")){//keszpenzes fizetes
              keszpenzesFizetes=true;
              visszajaro = Integer.parseInt(args[6]);
              System.out.println("Visszajaro:(szam) "+visszajaro);
       } else {//bankkartyas
              keszpenzesFizetes=false;
              kartyaszam = args[6];
              System.out.println("Kartyaszam:(szoveg) " + kartyaszam);
       }
}

/** Fizetesi adatok listazasa a fizetes modjanak megfeleloen.
 */
/*@
@public normal_behavior
@      requires keszpenzesFizetes;
@      assignable \nothing;
@      ensures \result ==
@      "Bizonylat azontosito: "+ String.valueOf(bizonylatAzonosito)+
@       "; Fizetendo osszeg: " + String.valueOf(fizetendoOsszeg)+
@      "; Visszajaro: " + String.valueOf(visszajaro);
@also
@public normal_behavior
@      requires !keszpenzesFizetes;
@      assignable \nothing;
@      ensures \result ==
@      "Bizonylat azonosito: "+String.valueOf(bizonylatAzonosito)+
@      "; Fizetendo osszeg: " + String.valueOf(fizetendoOsszeg)+
@      "; Kartyaszam: " + String.valueOf(kartyaszam);
@*/
public /*@ pure @*/ String getFizetesInfo(String[] args){
       if(keszpenzesFizetes) //keszpenzes fizetes
       return "Bizonylat azonosito: "+
              String.valueOf(bizonylatAzonosito)+
              "; Fizetendo osszeg: "+ String.valueOf(fizetendoOsszeg)+
              "; Visszajaro: "+String.valueOf(visszajaro);
       else //bankkartyas fizetes
       return "Bizonylat azonosito: "+
              String.valueOf(bizonylatAzonosito)+
              "; Fizetendo osszeg: "+ String.valueOf(fizetendoOsszeg)+
```

```
                "; Kartyaszam: "+ String.valueOf(kartyaszam);
        }
}
```

A „*Fizetés*” döntéssel kapcsolatos két ekvivalens döntéspredikátumú döntést találunk, amelyek összevonása indokolt lehet. („3.2 A döntésredundanciák elkerülésének esetei”, „4.4 Döntésösszevonás eseteinek vizsgálata JML-lel”)
A fizetéssel kapcsolatos első döntés arról határoz, hogy a fizetés módjának megfelelően milyen adatokra van szükség:

```
/**
 *  Megkapja a fizetendo osszeget es a bizonylat szamot,
 *  amelyek a fizetes modjatol fuggetlen adatok.
 *  A visszajaro es a kartyaszam a fizetes modjatol fuggo adatok.
 */
/*@
@public normal_behavior
@      requires args[3]!="nem" &&
@             parseable(args[4])&&parseable(args[5])&&
@             parseable(args[6]);
@      assignable keszpenzesFizetes, bizonylatAzonosito,
@             fizetendoOsszeg, visszajaro, System.out;
@      ensures keszpenzesFizetes == true &&
@             bizonylatAzonosito == Integer.parseInt(args[4])&&
@             fizetendoOsszeg == Integer.parseInt(args[5])&&
@             visszajaro == Integer.parseInt(args[6]);
@also
@public normal_behavior
@      requires args[3]=="nem" &&
@             parseable(args[4])&&parseable(args[5])&&
@                    parseable(args[6]);
@      assignable keszpenzesFizetes, bizonylatAzonosito,
@                           fizetendoOsszeg,kartyaszam, System.out;
@      ensures keszpenzesFizetes == false &&
@             bizonylatAzonosito == Integer.parseInt(args[4]) &&
@             fizetendoOsszeg ==Integer.parseInt(args[5])&&
@                           kartyaszam == args[6];
@also
@public exceptional_behavior
@      requires !parseable(args[4])||!parseable(args[5])||
@                    !parseable(args[6]);
@      assignable \nothing;
@      signals_only java.lang.NumberFormatException;
@*/
public void setFizetesInfo(String[] args) throws
                                  NumberFormatException {…}
```

A fizetés módjának megfelelően állítjuk be a fizetéssel kapcsolatos adatokat, mivel készpénzes és bankkártyás fizetés esetén részben más adatokat kell bekérni. A döntéspredikátum készpénzes vagy bankkártyás fizetés esetén a következő:

```
      args[3]!="nem",args[3]=="nem"
```

Ezzel a másik döntés döntéspredikátuma ekvivalens, mivel döntéspredikátuma a *keszpenzesFizetés* logikai változótól függ, amit az első fizetés módjával kapcsolatos döntés döntéslehetőségeiben állítunk be:

```
/** Fizetesi adatok listazasa a fizetes modjanak megfeleloen.
 */
/*@
@public normal_behavior
@      requires keszpenzesFizetes;
@      assignable \nothing;
@      ensures \result ==
@      "Bizonylat azontosito: "+ String.valueOf(bizonylatAzonosito)+
@      "; Fizetendo osszeg: " + String.valueOf(fizetendoOsszeg)+
@      "; Visszajaro: " + String.valueOf(visszajaro);
@also
@public normal_behavior
@      requires !keszpenzesFizetes;
@      assignable \nothing;
@      ensures \result ==
@      "Bizonylat azonosito: "+ String.valueOf(bizonylatAzonosito)+
@      "; Fizetendo osszeg: " + String.valueOf(fizetendoOsszeg)+
@      "; Kartyaszam: " + String.valueOf(kartyaszam);
@*/
public /*@ pure @*/ String getFizetesInfo(String[] args){…}
```

Mivel a két fizetés módjával kapcsolatos döntés döntéspredikátuma ekvivalens, az összevonásuk-kiemelésük indokolt.

Észre kell vennünk, hogy a példában szereplő döntések felépítése a „*Híd*” tervezési minta döntésszerkezetével egyezik. A „*Vásárlás”* döntés a „*Híd”* tervezési minta „*Elvont ábrázolás”* döntésének felel meg, a „*Fizetés”* döntés a minta „*Megvalósítás”* döntéseként értelmezhető. Látható, hogy a „*Vásárlás” – „Elvont ábrázolás”* döntés döntéslehetőségeit reprezentáló Vasarlas.*getBizonylat(...) – ElvontÁbrázolás.művelet()* metódus hogyan tartalmazza a „*Fizetés” – „Megvalósítás”* döntést, amit a *Fizetés.getFizetesInfo(...)-Megvalósítás.műveletMegvalósítás()* mellékhatásmentes metódushívással érhetünk el.

A döntések kiemelésével, összevonásával kialakítható a „*Híd*” tervezési minta szerinti optimálisabb szerkezet, ahol a döntéseket már osztályhierarchiába ábrázoljuk. Ezek szerint tehát a „*Vásárlás”* és a „*Fizetés”* döntések összevonása és kiemelése után a „*Híd*” tervezési minta osztályszerkezete elérhető.
A döntésszerkezet alapján tehát a „Híd” tervezési mintánál ismert osztályszerkezet [15] kialakításával építhető ki az optimális osztály- és objektum szerkezet, melyet a döntések osztályhierarchiába történő kiemelésével érhetünk el.

5.4.2.2 Híd tervezési minta tulajdonságai döntéskiemelés után

A *Vásárlás* és a *Fizetés* döntések összevonása és kiemelése után a döntések külön osztályhierarchiában reprezentáltak.

```
package hu.dontes.szint3;
import hu.dontes.szint3.KeszpenzesFizetes;
import hu.dontes.szint3.BankkartyasFizetes;
import hu.dontes.szint3.Fizetes;
//@ model import org.jmlspecs.models.*;
```

```
/** Vasarlasi es fizetesi adatok bekerese, vasarlas es fizetes
modjanak megfelelo bizonylat nyomtatasa.
*/
public abstract class Vasarlas {
/**    Vasarlas tipusanak modellvaltozoja: */
//@ public model instance boolean m_szemelyesVasarlas;

/**    Szemelyes vasarlas modell valtozoi: */
//@ public model instance String m_vezeteknev;
//@ public instance initially m_vezeteknev == "";

//@ public model instance String m_keresztnev;
//@ public instance initially m_keresztnev == "";

/**    Ceges vasarlas modell valtozoi: */
//@ public model instance String m_cegNev;
//@ public instance initially m_cegNev == "";

//@ public model instance String m_cegSzekhely;
//@ public instance initially m_cegSzekhely == "";

/**    Modell fuggvennyel vizsgaljuk az bejovo argumentumokat: */
/*@    public static pure model boolean parseable( String s ) {
@      try { int d = Integer.parseInt(s); return true; }
@      catch (Exception e) { return false; }}
@*/

/**    A Fizetes osztaly alosztalyai a KeszpenzesFizetes vagy
 * BankkartyasFizetes osztalyok egyede hasznalhato a fizetesi
 * feltetelek kezelesere.
 */
private /*@ spec_public @*/ Fizetes     fizetes;

/** Vasarlas lehet szemelyes vagy ceges vasarlas, aminek megfeleloen
 *  kulonbozo adatokat kell bekerni es a bizonylatra nyomtatni.
 */
/*@
@public normal_behavior
@      requires args[0] != "nem";
@      assignable System.out;
@      ensures \result instanceof SzemelyesVasarlas &&
@                    (\result).m_szemelyesVasarlas == true;
@ also
@      requires args[0] == "nem";
@      assignable System.out;
@      ensures \result instanceof CegesVasarlas &&
@                    (\result).m_szemelyesVasarlas == false;
@*/
public static Vasarlas getVasarlas(String[] args){
       System.out.println("Szemelyes vasarlas?:(igen/nem) "+args[0]);
       /** Vasarlas tipusanak archivalasa.*/
       if(!args[0].equals("nem"))
              return new SzemelyesVasarlas();
       else
              return new CegesVasarlas();
       }
```

```
public static void main(String[] args) {
        //Ha az argumentum szam nem megfelelo.
        if(args.length < 7 ){
                System.err.println("Az argumentum szam nem megfelelo!");
                System.exit(-1);
        }
        try{
                //Vasarlas tipusanak beallitasa.
                Vasarlas vasarlas=Vasarlas.getVasarlas(args);
                //Fizetesi adatok megadasa.
                vasarlas.setFizetes(args);
                //Vasarlasi adatok megadasa.
                vasarlas.setVasarlasInfo(args);
                //Bizonylat nyomtatasa.
                System.out.println(vasarlas.getBizonylat());
                }
        catch (java.lang.NumberFormatException nfe){
                System.err.println("Az argumentumok formatuma nem " +
                                        "megfelelo!");
                System.exit(-1);
        }
}

/**     Vasarlasi adatok vasarlas tipusanak megfelelo beallitasa.
 */
/*@
@public normal_behavior
@       requires m_szemelyesVasarlas;
@       assignable m_vezeteknev, m_keresztnev;
@       ensures m_vezeteknev==args[1] && m_keresztnev==args[2];
@also
@public normal_behavior
@       requires !m_szemelyesVasarlas;
@       assignable m_cegNev, m_cegSzekhely;
@       ensures m_cegNev==args[1] && m_cegSzekhely==args[2];
@*/
public abstract void setVasarlasInfo(String[] args);

/**     Fizetes modjanak meghatarozasa, a fizetes modjanak megfelelo
 *      adatok bekerese.
 */
/*@
@public normal_behavior
@       requires args[3]!="nem" &&
@               parseable(args[4])&& parseable(args[5]) &&
@               parseable(args[6]);
@       assignable fizetes,fizetes.m_keszpenzesFizetes,
@               fizetes.bizonylatAzonosito,fizetes.fizetendoOsszeg,
@               fizetes.m_visszajaro, System.out;
@       ensures fizetes instanceof KeszpenzesFizetes &&
@               fizetes.m_keszpenzesFizetes == true &&
@               fizetes.bizonylatAzonosito==Integer.parseInt(args[4])&&
@               fizetes.fizetendoOsszeg==Integer.parseInt(args[5])&&
@               fizetes.m_visszajaro==Integer.parseInt(args[6]);
@also
@public normal_behavior
@       requires args[3]=="nem" && parseable(args[4])&&
```

```
@            parseable(args[5]) && parseable(args[6]);
@      assignable fizetes,fizetes.m_keszpenzesFizetes,
@            fizetes.bizonylatAzonosito, fizetes.fizetendoOsszeg,
@            fizetes.m_kartyaszam, System.out;
@      ensures fizetes instanceof BankkartyasFizetes &&
@            fizetes.m_keszpenzesFizetes == false &&
@            fizetes.bizonylatAzonosito== Integer.parseInt(args[4])&&
@            fizetes.fizetendoOsszeg == Integer.parseInt(args[5])&&
@            fizetes.m_kartyaszam == args[6];
@also
@public exceptional_behavior
@      requires !parseable(args[4])|||!parseable(args[5])||
@            !parseable(args[6]);
@      assignable \nothing;
@      signals_only java.lang.NumberFormatException;
@*/
public void setFizetes(String[] args)
                    throws java.lang.NumberFormatException{
      System.out.println("Keszpenzes fizetes?:(igen/nem) "+args[3]);
      //Fizetes modjanak rogzitese Fizetes tipusu referenciaval.
      if(!args[3].equals("nem"))
            fizetes=new KeszpenzesFizetes();
      else
            fizetes=new BankkartyasFizetes();
      fizetes.setFizetesInfo(args);
}

/** Fizetes modjanak megfelelo referencia elkerese.
 */
/*@
@public normal_behavior
@      assignable \nothing;
@      ensures \result == fizetes;
@*/
public /*@ pure @*/ Fizetes getFizetes(){
      return fizetes;
}

/**
 * Bizonylat (vasarlasi es fizetesi adatok) kiiratasa.
 */
/*@
@public normal_behavior
@      requires m_szemelyesVasarlas;
@      assignable \nothing;
@      ensures \result == "Vezeteknev: " + m_vezeteknev +
@            "; Keresztnev: " + m_keresztnev +
@            "\nFizetesi adatok: " + getFizetes().getFizetesInfo();
@also
@public normal_behavior
@      requires !m_szemelyesVasarlas;
@      assignable \nothing;
@      ensures \result == "Ceg neve: " + m_cegNev +
@            "; Ceg cime: " + m_cegSzekhely+
@            "\nFizetesi adatok: " + getFizetes().getFizetesInfo();
@*/
public abstract String getBizonylat();
```

```
}
```

A „*Vasarlas*” osztály a „*Vásárlás*” döntés osztályhierarchia szülőosztálya, ahol a döntés nem döntés specifikus részének keretét deklaráljuk. A döntéslehetőségek viselkedési specifikációja is megjelenik a „*Vásárlás*” osztály JML alapú viselkedés specifikációban, ahol a döntéslehetőség specifikus változókat modellváltozók képviselik.

```
/**    Szemelyes vasarlas modell valtozoi: */
//@ public model instance String m_vezeteknev;
//@ public instance initially m_vezeteknev == "";

//@ public model instance String m_keresztnev;
//@ public instance initially m_keresztnev == "";

/**    Ceges vasarlas modell valtozoi: */
//@ public model instance String m_cegNev;
//@ public instance initially m_cegNev == "";

//@ public model instance String m_cegSzekhely;
//@ public instance initially m_cegSzekhely == "";
```

A döntéslehetőségek szerinti alosztályok példányosítása a *Vasarlas.getVasarlas(String[] args)* metódusban történik, ahol a példányosításnak megfelelően kap értéket a modell szintű döntéspredikátum. A döntéspredikátum az egyes alosztályokban a döntéslehetőségnek megfelelő skaláris logikai értéket kap. (Lásd lentebb!)
Az ekvivalens döntések kiemelése és összevonása után a következő módokon polimorf metódusokban jelennek meg:

```
/**    Vasarlasi adatok vasarlas tipusanak megfelelo beallitasa.
 */
/*@
@public normal_behavior
@      requires m_szemelyesVasarlas;
@      assignable m_vezeteknev, m_keresztnev;
@      ensures m_vezeteknev==args[1] && m_keresztnev==args[2];
@also
@public normal_behavior
@      requires !m_szemelyesVasarlas;
@      assignable m_cegNev, m_cegSzekhely;
@      ensures m_cegNev==args[1] && m_cegSzekhely==args[2];
@*/
public abstract void setVasarlasInfo(String[] args);

/**
 * Bizonylat (vasarlasi es fizetesi adatok) kiiratasa.
 */
/*@
@public normal_behavior
@      requires m_szemelyesVasarlas;
@      assignable \nothing;
@      ensures \result == "Vezeteknev: " + m_vezeteknev +
@            "; Keresztnev: " + m_keresztnev +
@            "\nFizetesi adatok: " + getFizetes().getFizetesInfo();
```

```
@also
@public normal_behavior
@      requires !m_szemelyesVasarlas;
@      assignable \nothing;
@      ensures \result == "Ceg neve: " + m_cegNev +
@             "; Ceg cime: " + m_cegSzekhely+
@             "\nFizetesi adatok: " + getFizetes().getFizetesInfo();
@*/
public abstract String getBizonylat();
```

Látnunk kell, hogy az absztrakt metódusok JML alapú viselkedés specifikációja hogyan ábrázolja a polimorf működés lehetőségeit, azaz a döntéslehetőség specifikus viselkedéseket.
A mellékhatásmentes „*Fizetés*” döntés a *getFizetes().getFizetesInfo()* metódushívással jelenik meg a *getBizonylat()* metódusban. Mivel a metódus mellékhatásmentes, ezért erre a JML alapú viselkedés specifikációban lehetőség van.

A „*Vásárlás*” döntés döntéslehetőségeit a következő alosztályok adják meg:

```
package hu.dontes.szint3;
/**
 * A Vasarlas osztaly alosztalyakent szemelyes vasarlas eseten
hasznalatos.
 * Szemelyes vasarlas adatainak bekerese, megfelelo bizonylat
nyomtatasa.
 */
public class SzemelyesVasarlas extends Vasarlas{

private /*@ spec_public @*/ String keresztnev = "";
//@ public instance initially keresztnev == "";
private /*@ spec_public @*/ String vezeteknev = "";
//@ public instance initially vezeteknev == "";

/**    Modell valtozokhoz reprezentaciot rendelunk: */
//@ public instance represents m_szemelyesVasarlas <- true;

//@ public instance represents m_keresztnev <- keresztnev;
//@ public instance represents m_vezeteknev <- vezeteknev;

//@ public instance represents m_cegNev <- null;
//@ public instance represents m_cegSzekhely <- null;

/**    Szemelyes vasarlasi adatok beallitasa.
 */
/*@
@also
@public normal_behavior
@      assignable vezeteknev, keresztnev;
@      ensures vezeteknev == args[1] && keresztnev == args[2];
@*/
public void setVasarlasInfo(String[] args){
       vezeteknev = args[1];
       System.out.println("Vezeteknev: " + vezeteknev);
       keresztnev = args[2];
       System.out.println("Keresztnev: " + keresztnev);
```

```
}

/**
 * Bizonylat (szemelyes vasarlasi es fizetesi adatok) kiiratasa.
 */
/*@ also
@  public normal_behavior
@      assignable \nothing;
@      ensures \result == "Vezeteknev: " + vezeteknev +
@      "; Keresztnev: " + keresztnev +
@      "\nFizetesi adatok: " + getFizetes().getFizetesInfo();
@*/
public String getBizonylat(){
       return "Bizonylat: \n"+
              "Vezeteknev: "+ vezeteknev +
              "; Keresztnev: " + keresztnev+
              "\nFizetesi adatok: " + getFizetes().getFizetesInfo();
       }
}

package hu.dontes.szint3;
/**
 *  A Vasarlas osztaly alosztalyakent ceges vasarlas eseten
hasznalatos.
 *  Ceges vasarlasi adatok bekerese, megfelelo bizonylat nyomtatasa.
 */
public class CegesVasarlas extends Vasarlas{

private /*@ spec_public @*/ String cegNev = "";
//@ public instance initially cegNev == "";
private /*@ spec_public @*/ String cegSzekhely = "";
//@ public instance initially cegSzekhely == "";

/**    Modell valtozokhoz reprezentaciot rendelunk: */
//@ public instance represents m_szemelyesVasarlas <- false;

//@ public instance represents m_keresztnev <- null;
//@ public instance represents m_vezeteknev <- null;

//@ public instance represents m_cegNev <- cegNev;
//@ public instance represents m_cegSzekhely <- cegSzekhely;

/**
 * Ceges vasarlasi adatok beallitasa.
 */
/*@
@also
@public normal_behavior
@      assignable cegNev, cegSzekhely;
@      ensures cegNev == args[1] && cegSzekhely == args[2];
@*/
public void setVasarlasInfo(String[] args){
       cegNev = args[1];
       System.out.println("Ceg neve: " + cegNev);
       cegSzekhely = args[2];
       System.out.println("Ceg szekhelye: " + cegSzekhely);
       }
```

```
/**
 * Bizonylat (ceges vasarlasi es fizetesi adatok) kiiratasa.
 */
/*@
@also
@public normal_behavior
@      assignable \nothing;
@      ensures \result == "Ceg neve: " + cegNev +
@      "; Ceg cime: " + cegSzekhely+
@      "\nFizetesi adatok: " + getFizetes().getFizetesInfo();
@*/
public String getBizonylat(){
       return "Bizonylat: \n"+
              "Ceg neve: "+ cegNev + "; Ceg szekhelye: "+ cegSzekhely+
              "\nFizetesi adatok: " + getFizetes().getFizetesInfo();
       }
}
```

A szülőosztályban megadott döntéslehetőség specifikus modellváltozók már implementációs változókkal reprezentáltak, melyek között a megfeleltetéseket az alosztályok JML specifikációjában el kell végezni:
Személyes vásárlás esetén:

```
       /**    Modell valtozokhoz reprezentaciot rendelunk: */
       //@ public instance represents m_szemelyesVasarlas <- true;

       //@ public instance represents m_keresztnev <- keresztnev;
       //@ public instance represents m_vezeteknev <- vezeteknev;

       //@ public instance represents m_cegNev <- null;
       //@ public instance represents m_cegSzekhely <- null;
```

Céges vásárlás esetén:

```
       /**    Modell valtozokhoz reprezentaciot rendelunk: */
       //@ public instance represents m_szemelyesVasarlas <- false;

       //@ public instance represents m_keresztnev <- null;
       //@ public instance represents m_vezeteknev <- null;

       //@ public instance represents m_cegNev <- cegNev;
       //@ public instance represents m_cegSzekhely <- cegSzekhely;
```

A „*Vásárlás*” döntés döntéspredikátuma skaláris logikai értéket kap a megfelelő döntéslehetőség szerinti alosztály példányosításakor, illetve a döntéslehetőség szerint kötjük össze a modellváltozókat az implementációs változókkal.

Az egyes alosztályok a szülőosztályban megadott döntéseknek csak a döntéslehetőségnek megfelelő szeletét valósítják meg, melynek megfelelően a JML alapú viselkedés specifikációban is csak a döntéslehetőség szerinti viselkedést találjuk, ahol már a modell változókat helyettesítő implementációs változók jelennek meg.

A „*Fizetes*” osztály a „*Fizetés*” döntés osztályhierarchia szülőosztálya, ahol a döntés nem döntés specifikus részének keretét deklaráljuk.

```
package hu.dontes.szint3;
//@ model import org.jmlspecs.models.*;
/**
 *     Fizetes modjat meghatarozo osztalyok absztrakt szuloosztalya,
 * amely a fizetes modjainak kozos feluletet szablya meg.
 * Fizetes modja lehet keszpenz vagy bankkartya alapu, amely szerint
 * mas adatok bekerese, nyomtatasa szukseges.
 */
public abstract class Fizetes {
/**    Fizetes modjanak modellvaltozoja: */
//@ public model instance boolean m_keszpenzesFizetes;

/**    Keszpenzes fizetes modell valtozoi: */
//@ public model instance int m_visszajaro;
//@ public instance initially m_visszajaro == 0;

/**    Bankkartyas fizetes modell valtozoi: */
//@ public model instance String m_kartyaszam;
//@ public instance initially m_kartyaszam == "";

/**    Modell fuggvennyel vizsgaljuk az bejovo argumentumokat: */
/*@    public static pure model boolean parseable( String s ) {
@      try { int d = Integer.parseInt(s); return true; }
@      catch (Exception e) { return false; }}
@*/

/**    Keszpenzes es bankkartyas fizetes eseten is szukseges
megadni:*/
protected /*@ spec_public @*/ long bizonylatAzonosito = 0;
//@ public instance initially bizonylatAzonosito == 0;
protected /*@ spec_public @*/ int fizetendoOsszeg=0;
//@ public instance initially fizetendoOsszeg == 0;

/**
 *  Megkapja a fizetendo osszeget es a bizonylat szamot, amelyek
 *  a fizetes modjatol fuggetlen adatok.
 *  A visszajaro es a kartyaszam viszont fizetes modjatol fuggo
 *  adatok.
 */
/*@
@public normal_behavior
@      requires m_keszpenzesFizetes &&
@             parseable(args[4]) && parseable(args[5]) &&
@             parseable(args[6]);
@      assignable bizonylatAzonosito, fizetendoOsszeg, System.out;
@      ensures bizonylatAzonosito == Integer.parseInt(args[4])&&
@             fizetendoOsszeg == Integer.parseInt(args[5]) &&
@             m_visszajaro == Integer.parseInt(args[6]);
@also
@public normal_behavior
@      requires !m_keszpenzesFizetes &&
@             parseable(args[4])&&parseable(args[5])&&
@             parseable(args[6]);
@      assignable bizonylatAzonosito,fizetendoOsszeg,System.out;
@      ensures bizonylatAzonosito == Integer.parseInt(args[4])&&
@             fizetendoOsszeg == Integer.parseInt(args[5]) &&
```

```
@            m_kartyaszam == args[6];
@also
@      public exceptional_behavior
@      requires !parseable(args[4])||!parseable(args[5])||
@            !parseable(args[6]);
@      assignable \nothing;
@      signals_only java.lang.NumberFormatException;
@*/
public void setFizetesInfo(String[] args) throws
                                        NumberFormatException{
       //Keszpenzes es bankkartyas fizetes eseten is szukseges
       adatok:
       bizonylatAzonosito = Integer.parseInt(args[4]);
       System.out.println("Bizonylat Azonosito:(Szam) "
                                          +bizonylatAzonosito);
       fizetendoOsszeg = Integer.parseInt(args[5]);
       System.out.println("Fizetendo osszeg:(Szam)
       "+fizetendoOsszeg);
       }

/** Fizetesi adatok listazasa a fizetes modjanak megfeleloen.*/
/*@
@public normal_behavior
@      requires m_keszpenzesFizetes;
@      assignable \nothing;
@      ensures \result ==
@      "Bizonylat azontosito: "+ String.valueOf(bizonylatAzonosito)+
@      "; Fizetendo osszeg: "+String.valueOf(fizetendoOsszeg)+
@      "; Visszajaro: "+String.valueOf(m_visszajaro);
@also
@public normal_behavior
@      requires !m_keszpenzesFizetes;
@      assignable \nothing;
@      ensures \result ==
@      "Bizonylatazonosito:"+String.valueOf(bizonylatAzonosito)+
@      "; Fizetendo osszeg: "+String.valueOf(fizetendoOsszeg)+
@      "; Kartyaszam: "+String.valueOf(m_kartyaszam);
@*/
public abstract /*@ pure @*/ String getFizetesInfo();

}
```

A „*Fizetés*” döntés JML alapú viselkedés specifikációban a döntéslehetőségek viselkedését is specifikáljuk, ahol a döntéslehetőség specifikus változókat modell változók képviselik.

```
       /**    Keszpenzes fizetes modell valtozoi: */
       //@ public model instance int m_visszajaro;
       //@ public instance initially m_visszajaro == 0;

       /**    Bankkartyas fizetes modell valtozoi: */
       //@ public model instance String m_kartyaszam;
       //@ public instance initially m_kartyaszam == "";
```

A „*Fizetés*” döntés megfelelő döntéslehetősége szerinti alosztályok példányosítása a *Vasarlas.setFizetes(String[] args)* metódusban történik, ahol a modell szintű döntéspredikátum a példányosításnak megfelelően kap értéket. A döntéspredikátum az egyes alosztályokban a döntéslehetőségnek megfelelő skaláris logikai értéket kap. (Lásd az „*KeszpenzesFizetes*”, „*BankkartyasFizetes*” alosztályok!)

Az összevont ekvivalens döntéspredikátumú döntések egyikét a *Fizetes.setFizetesInfo(...)* polimorf metódus képviseli, a másikat a *Fizetes.getFizetesInfo()* absztrakt polimorf metódusban találjuk, melyek JML alapú viselkedés specifikációja a következő:

```
/**
 *  Megkapja a fizetendo osszeget es a bizonylat szamot, amelyek
 *  a fizetes modjatol fuggetlen adatok.
 *  A visszajaro es a kartyaszam viszont fizetes modjatol fuggo
 *  adatok.
 */
/*@
@ public normal_behavior
@      requires m_keszpenzesFizetes &&
@            parseable(args[4])&&parseable(args[5])&&
@            parseable(args[6]);
@      assignable bizonylatAzonosito,fizetendoOsszeg, System.out;
@      ensures bizonylatAzonosito== Integer.parseInt(args[4])&&
@            fizetendoOsszeg == Integer.parseInt(args[5])&&
@            m_visszajaro==Integer.parseInt(args[6]);
@also
@public normal_behavior
@      requires !m_keszpenzesFizetes && parseable(args[4])&&
@            parseable(args[5])&& parseable(args[6]);
@      assignable bizonylatAzonosito,fizetendoOsszeg,System.out;
@      ensures bizonylatAzonosito == Integer.parseInt(args[4])&&
@              fizetendoOsszeg == Integer.parseInt(args[5]) &&
@            m_kartyaszam == args[6];
@also
@public exceptional_behavior
@      requires !parseable(args[4])||!parseable(args[5])||
@            !parseable(args[6]);
@      assignable \nothing;
@      signals_only java.lang.NumberFormatException;
@*/
public void setFizetesInfo(String[] args) throws
                    NumberFormatException {…}

/** Fizetesi adatok listazasa a fizetes modjanak megfeleloen.*/
/*@
@public normal_behavior
@      requires m_keszpenzesFizetes;
@      assignable \nothing;
@      ensures \result ==
@      "Bizonylat azontosito: "+ String.valueOf(bizonylatAzonosito)+
@            "; Fizetendo osszeg: "+String.valueOf(fizetendoOsszeg)+
@            "; Visszajaro: "+String.valueOf(m_visszajaro);
@      also
```

```
@      public normal_behavior
@             requires !m_keszpenzesFizetes;
@             assignable \nothing;
@             ensures \result ==
@             "Bizonylatazonosito:"+String.valueOf(bizonylatAzonosito)
@             +"; Fizetendo osszeg:"+String.valueOf(fizetendoOsszeg)+
@             "; Kartyaszam: "+String.valueOf(m_kartyaszam);
@*/
public abstract /*@ pure @*/ String getFizetesInfo();
```

A *setFizetesInfo(...)* metódus nem absztrakt, azaz tartalmaz döntés független viselkedést, amit a polimorf metódus szülőosztályban megadott változatában deklarálunk. A metódus JML-es viselkedés specifikációja jól mutatja, hogyan egészül ki a metódus viselkedése a döntéslehetőség-specifikus részekkel.
A *getFizetesInfo()* absztrakt metódus esetén nincs olyan viselkedés, amely döntés független, azaz mindkét döntéslehetőség esetén érvényesül. A JML alapú specifikációban a polimorf viselkedések lehetőségei jól láthatók.

A „*Vásárlás*” döntés döntéslehetőségeit a következő alosztályok adják meg:

```
package hu.dontes.szint3;
//@ model import org.jmlspecs.models.*;
import hu.dontes.szint3.Fizetes;
/**
 * A Fizetes osztaly alosztalyakent keszpenzes fizetes eseten
hasznalatos.
 * Keszpenzes fizetesi adatok bekerese, nyomtatasa.
 */
public class KeszpenzesFizetes extends Fizetes{

private /*@ spec_public @*/ int visszajaro = 0;
//@ public instance initially visszajaro == 0;

/**    Modell valtozokhoz reprezentaciot rendelunk: */
//@ public instance represents m_keszpenzesFizetes <- true;
//@ public instance represents m_visszajaro <- visszajaro;
//@ public instance represents m_kartyaszam <- null;

/**    Ha a vasarlo keszpenzzel fizet, akkor a visszajarot is
 *  meg kell adni.
 */
/*@ also
@      public behavior
@             requires parseable(args[6]);
@             assignable visszajaro ,System.out;
@             ensures visszajaro == Integer.parseInt(args[6]);
@      also
@      public exceptional_behavior
@             requires !parseable(args[6]);
@             assignable \nothing;
@             signals_only java.lang.NumberFormatException;
@*/
public void setFizetesInfo(String[] args)
                                   throws NumberFormatException{
```

```
        super.setFizetesInfo(args);
        visszajaro = Integer.parseInt(args[6]);
        System.out.println("Visszajaro:(Szam) " + visszajaro);
        }

/** Fizetesi adatok nyomtatasa a fizetes tipusanak megfeleloen.
 */
/*@ also
@public normal_behavior
@       assignable \nothing;
@       ensures \result ==
@       "Bizonylat azontosito: "+ String.valueOf(bizonylatAzonosito)+
@       "; Fizetendo osszeg: "+String.valueOf(fizetendoOsszeg)+
@       "; Visszajaro: "+String.valueOf(m_visszajaro);
@*/
public /*@ pure @*/ String getFizetesInfo(){
        return "Bizonylat azonosito: "+
                        String.valueOf(bizonylatAzonosito)+
                    "; Fizetendo osszeg: "+
                        String.valueOf(fizetendoOsszeg)+
                    "; Visszajaro: "+String.valueOf(visszajaro);
        }
}

package hu.dontes.szint3;
//@ model import org.jmlspecs.models.*;
/**
 * A Fizetes osztaly alosztalyakent bankkartyas fizetes eseten
hasznalatos.
 * Bankkartyas fizetesi adatok bekerese, nyomtatasa.
 */
public class BankkartyasFizetes extends Fizetes{

private /*@ spec_public @*/ String kartyaszam = "";
//@ public instance initially m_kartyaszam == "";

/**    Modell valtozokhoz reprezentaciot rendelunk: */
//@ public instance represents m_keszpenzesFizetes <- true;
//@ public instance represents m_kartyaszam <- kartyaszam;
//@ public instance represents m_visszajaro <- 0;

/**    Ha a vasarlo bankkartyaval fizet, akkor a kartyaszamot
 * is meg kell adni.
 */
/*@
@also
@public behavior
@       requires parseable(args[6]);
@       assignable kartyaszam, System.out;
@       ensures kartyaszam == args[6];
@*/
public void setFizetesInfo(String[] args){
        super.setFizetesInfo(args);
        kartyaszam = args[6];
        System.out.println("Kartyszam:(szoveges)" + kartyaszam);

        }
```

```
/** Fizetesi adatok nyomtatasa fizetes tipusanak megfeleloen.
 */
/*@ also
@public normal_behavior
@      assignable \nothing;
@      ensures \result ==
@      "Bizonylat azonosito: "+ String.valueOf(bizonylatAzonosito)+
@      "; Fizetendo osszeg:" + String.valueOf(fizetendoOsszeg)+
@      "; Kartyaszam: "+String.valueOf(m_kartyaszam);
@*/
public /*@ pure @*/ String getFizetesInfo(){
       return "Bizonylat azonosito: "
                                  +String.valueOf(bizonylatAzonosito)+
              "; Fizetendo osszeg: "+String.valueOf(fizetendoOsszeg)+
                                  "; Kartyaszam: " + kartyaszam;
       }
}
```

A szülőosztályban megadott döntéslehetőség specifikus modellváltozók már implementációs változókkal reprezentáltak, melyek között a megfeleltetéseket is el kell végezni az alosztályok JML specifikációjában.
Készpénzes fizetés esetén:

```
       /**    Modell valtozokhoz reprezentaciot rendelunk: */
       //@ public instance represents m_keszpenzesFizetes <- true;
       //@ public instance represents m_visszajaro <- visszajaro;
       //@ public instance represents m_kartyaszam <- null;
```

Bankkártyás fizetés esetén:

```
       /**    Modell valtozokhoz reprezentaciot rendelunk: */
       //@ public instance represents m_keszpenzesFizetes <- true;
       //@ public instance represents m_kartyaszam <- kartyaszam;
       //@ public instance represents m_visszajaro <- 0;
```

A „*Fizetés*” döntés döntéspredikátuma skaláris logikai értéket kap a megfelelő döntéslehetőség szerinti alosztály példányosításakor, illetve a döntéslehetőség szerint kötjük össze a modellváltozókat az implementációs változókkal.

Az egyes alosztályok a szülőosztályban megadott döntéseknek csak a döntéslehetőségnek megfelelő szeletét valósítják meg, melynek megfelelően a JML alapú viselkedés specifikációban is csak a döntéslehetőség szerinti viselkedést találjuk, ahol már a modell változókat helyettesítő implementációs változók jelennek meg.

5.4.3 Tervezési minták döntésszerkezetek szerinti JML alapú formalizálásának szabályai

A tervezési mintákat a bennük lévő döntések tulajdonságai alapján jellemezhetjük.
A döntésszerkezetetek a JML alapú viselkedés specifikációban is megjelennek úgy, hogy az előzőekben ismertetett módon a döntések eset specifikus tartalmától függetlenül megadható az a „váz”, ami az egyes tervezési mintákra jellemző.

Látni kell, hogy a példában is szereplő döntésszerkezetek szerinti döntési váz csak akkor ismerhető fel, ha meg tudjuk állapítani, hogy a viselkedés specifikációk döntéspredikátumai, utófeltételei ekvivalensek-e vagy sem.

A következőkben a döntések JML alapú viselkedés specifikációjának jellemzőit összegezzük, melyek a tervezési minta specifikálás szempontjából is érdekesek („4. Döntések vizsgálata JML-lel" fejezet):

- Másként épülnek fel a mellékhatásmentes – nem mellékhatásmentes metódusokkal reprezentált döntések viselkedés specifikációi.
    - Mellékhatásmentes metódus esetén:
        - Mellékhatásmentes metódusokkal ábrázolható döntések esetén nem történik állapotátmenet, vagyis az adat struktúra egyetlen változójának sem változik az értéke. Ezt az *assignable \nothing* kifejezéssel jelezzük az adott metódus JML alapú viselkedési specifikációban.
        - A mellékhatásmentes metódusokat *pure* kulcsszóval deklaráljuk
        - A mellékhatásmentes metódussal reprezentált döntés esetén, a felhasználási helyein (például egy másik döntés része) nem szükséges a metódus viselkedési specifikációját kifejteni, a metódusra a viselkedési specifikációban is lehet hivatkozni
    - Mellékhatásos metódus esetén:
        - Történik állapotátmenet. Az *assignable* kifejezéssel kell megadni azon változókat, amelyek értéke változik.
        - A döntést reprezentáló metódus felhasználási helyein specifikálni kell a metódus viselkedését. Ha egy másik döntés részeként kerül meghívásra a metódus, akkor egy több szintű, összetett döntésszerkezet fog megjelenni a JML viselkedés specifikációban.
- Döntéspredikátum:
    - Nem kiemelt állapotban a döntéspredikátum az adatszerkezet részeként deklarált, amit az egyes döntésesetekben felhasználhatunk.
    - Döntések kiemelt állapota esetén a döntés reprezentáló osztálystruktúra szülőosztályában modellváltozóként deklaráljuk, amely az egyes alosztályokban kap az alosztálynak megfelelő döntéslehetőség specifikus értéket, azaz reprezentációja egy literálhoz kötéssel valósul meg.
- Döntések kiemelt állapota esetén a döntés reprezentáló osztály struktúra szülőosztályában modell változókkal kerülnek megadásra a döntéslehetőség specifikus változók.
- A kiemelt állapotú döntés osztálystruktúrájának alosztályaiban a szülőosztályban deklarált döntés specifikus modellváltozók összekapcsolódnak implementációs változókkal, amelyek az alosztályokban reprezentálják azokat.

6 Konklúzió

A dolgozatban felvázoltam – illetve a példával szemléltettem – az öröklődés értelmezésének kiterjesztését, miszerint annak célja a programban lévő döntések kiemelésével a döntésszám csökkentése, ami az ekvivalens döntések konzisztenciáját biztosítja megkönnyítve a karbantarthatóságot. Tisztáztam a **döntés, döntéslehetőségek, döntéspredikátum, döntéseset és döntéskiemelés** fogalmait, amelyre alapozva megfogalmaztam azon eseteket, amikor a döntésredundancia feloldás és döntéskiemelés indokolt lehet.

Az elképzelésre alapozva az objektumorientált tervezési alapelvekre és az objektumorientált alapeszközökre is új megközelítést adtam, amely könnyebbé teheti az objektumorientált módszertan értelmezését és alkalmazását. Ennek legfőbb eredménye, hogy a döntések elemzése és a felvázolt döntéskiemelés és döntésredundancia feloldás esetei meghatározzák azon helyzeteket, amikor az öröklődés, mint objektumorientált eszköz használata indokolt. Mivel az objektumorientált tervezési alapelvek egy része azzal foglalkozik, hogy milyen esetekben szükséges öröklődést vagy objektum összetételt használni, a kérdés újszerű megközelítése e problémák kezelésére jó megoldásokat adhat. A már meglévő alapelvek új értelmezése mellett megfogalmaztam két új alapelvet: „**Öröklést döntésredundancia megszüntetésre alkalmazzunk!**”, „**Döntésredundanciát kerüljük!**”

A program minőség mérése minden objektumorientált és nem objektumorientált rendszerben kritikus, mivel fontos, hogy a megfogalmazott tervezési és fejlesztési alapelvek teljesülését mérni tudjuk.

A jelenleg ismert objektumorientált programozási metrikákról áttekintést adtam, melyek a program minőség mérésére szempontok széles spektrumán adnak lehetőséget. A döntésszerkezet állapotának vizsgálatára alapozottan is bemutattam meglévő metrikákat, melyeket a dolgozatban ismertetett elképzelés szerint új metrikákkal egészítettem ki. A bevezetett „**Döntésabsztrakciós metrika”** a polimorf – nem polimorf döntésesetek arányát mutatja a programban, mellyel szemléltethető, hogy a program döntésszerkezet szerint mennyire optimalizált. Ehhez képest szofisztikáltabb képet kaphatunk a döntésszerkezetek állapotáról a többi javasolt metrika alkalmazásával, melyekhez viszont a döntések viselkedési szerződésének elemzése szükséges, amely használhatóságukat erősen korlátozza.

Annak érdekében, hogy azon helyzetek felderíthetők legyenek, hogy a döntéskiemelés, döntésredundancia feloldás mikor indokolt, szükséges egy formalizációs eszköz. Felvázoltam a program formalizációs eszközök céljait, ismertettem fejlődési irányukat, mellyel a Floyd féle induktív állítások módszerére [12] és Hoare féle deduktív levezetési rendszerére [17] alapozva kialakult a JML[8][9][25][26][27], a Java viselkedés specifikációs eszköz. A döntések formális leírására és elemzésére a JML-t használtam, mely szerint ismertettem az egyszintű és többszintű döntések nem kiemelt és kiemelt állapot szerinti JML specifikációjának tulajdonságait. JML-re alapozva is megfogalmaztam a korábban ismertetett döntésredundancia feloldás szabályait.

A program minőség, objektumorientált tervezési elvek teljesülése a tervezési minták használatával biztosítható, melyek alkalmazása jól strukturált, optimalizált

objektum és osztály szerkezetek létrejöttét eredményezi. Dolgozatomban bemutattam a tervezési minták célkitűzéseit, melyek természetesen átfedésben vannak a tervezési alapelvek céljaival. A tervezési minták általános céljait újszerű megközelítésben ismertettem, miszerint a tervezési minták célja a jól strukturáltság elérése, amit a döntésfüggőségek, mint egyfajta implementációs függőség feloldásával, azaz döntésismétlődések megszüntetésével érhetünk el. A tervezési minták a döntésredundanciák megszüntetésére adnak recepteket, vagyis arra, hogy a különböző strukturált döntési helyzetekben a döntésismétlődés megszüntetése miként biztosítható.

A tervezési minták ezen új megközelítésének egyik legfontosabb eredménye az, hogy a tervezési mintákat nem csak a jelenleg ismert osztály és objektum szerkezetük alapján kell felismernünk, hanem a döntések egymásba épülése, döntésszerkezete szerint is. Kijelenthető, hogy **a tervezési minták tervezési helyzet és nem osztályszerkezet függőek. A tervezési helyzeteket a döntési struktúrák konkrét tervezési szerkezettől függetlenül tudják ábrázolni**.

A jelenleg elterjedt tervezési minta gyűjtemény [15] mintáit döntés szerkezeteik szerint elemzem és megadom döntésstruktúra szerinti jellemzésüket. Ez lehetőséget ad arra, hogy az egyes tervezési minták szerinti tervezési helyzeteket a megadott döntésszerkezeti leírás alapján felismerjük és a minta alkalmazásával az optimalizált osztály és objektum szerkezetet kialakítsuk. Döntésszerkezeteik alapján a tervezési minták egy új osztályozását adtam meg.

Kitérek a tervezési minták - bennük lévő döntések állapota szerinti- jellemzésére. Érdekes, hogy a tervezési minták milyen formában jelennek meg, ha a döntésszerkezetük szerinti döntések nem kiemelt állapotban vannak. A „*Híd*” tervezési minta bemutatásával, döntésszerkezetének döntéskiemelés előtti és utáni állapotának jellemzésével szemléltettem az észrevételeket A „*Híd*” tervezési minta döntésszerkezetének állapota szerinti jellemzéseket egy példán keresztül mutattam be.

A tervezési minták JML alapú formális leírásai segítségével lehetőség van a döntésszerkezetek elemezésére aszerint, hogy viselkedésük elő- vagy/és utófeltételei részben vagy teljesen ekvivalensek-e. Ennek megfelelően lehet indokolt nem kiemelt döntések esetén a döntésredundancia feloldás a döntéskiemelés-összevonás alkalmazásával, mellyel a tervezési minták szerinti objektum és osztályszerkezet kialakulhat.

Célom, hogy a megfogalmazott alapelvek szerint részben a bevezetett Döntésabsztrakciós metrika felhasználásával empirikus adatokkal vizsgáljam a döntés struktúrák reprezentációját és a program minőség összefüggéseit.

Tervezem az ismertetett koncepcióra, illetve formalizmusra alapozva a döntésismétlődéseket feloldó döntésösszevonások, döntéskiemelések automatizálhatóságának lehetőségeinek vizsgálatát, amivel esetlegesen lehetővé tehető a tervezés, illetve megvalósítás minőségi javításának automatizálása is.

Vizsgálni tervezem, hogy a generikus programegységek a döntésszerkezetekre milyen hatással vannak, azaz értelmezhetők-e generikus osztályhierarchiák paraméterezett döntésekként.

Az elgondolás helyességének vizsgálataként az elmúlt másfél évben folyó eFilter kutatás-fejlesztéses projekt projektvezetőjeként és vezető tervezőjeként a tervezési

fázisában kezdeményeztem, hogy a tervezési minták alkalmazhatóságát a döntésszerkezetek elemzésével végezzük, mely során számos komplex tervezési problémára sikerült optimális megoldást találni. A projekt során a döntésszerkezetek – tervezési minták összefüggéseinek egy új irányát sikerült felfedezni, miszerint a tervezési mintákra jellemző döntési helyzetek már a használati eset modellekben is megjelennek.
Új kutatási irány ennek megfelelően az, hogy a tervezési minták a használati eset modellekben [40] miként jelennek meg, illetve hogy a tervezési minták döntésszerkezetei hogyan tükröződnek a használati esetek szintjén.

7 Összefogaló

Észrevételem szerint a tervezési alapelvek és tervezési minták legfontosabb célja az implementációs függőségek típusain belül a különböző döntésszerkezetek szerinti döntésredundanciák, mint egyfajta implementációs függőségek feloldása. Nem megfelelően implementált, redundáns döntések implementációs függőségeket eredményeznek, melyeket döntésszerkezeti implementációs függőségnek nevezek.
A programban lévő döntések során arról döntünk, hogy egy adott ponton milyen funkcionalitásra és/vagy adatszerkezetre van szükségünk. Amikor egy döntést definiálunk a döntés döntéslehetőségeinek funkcionalitását és/vagy adatszerkezetét adjuk meg. A döntéslehetőségen a döntés egy lehetséges kimenetelét értjük, amely a döntéslehetőségnek megfelelő funkcionalitás és/vagy adatstruktúra érvényre jutását jelenti, illetve a döntéslehetőség teljesülését jelentő döntéspredikátumot, ami alapján eldől, hogy egy adott döntési helyen melyik döntéslehetőség jut érvényre.
Ahhoz, hogy a döntésredundanciák feloldásának lehetőségeit megértsük az objektumorientált módszertan alapeszközének, az öröklődésnek az értelmezését kell kiterjeszteni: Az öröklődés a programban lévő döntésredundanciák feloldásának eszköze, amely a döntések összevonása és kiemelése után azok deklarációjára ad egy absztraktabb keretet. Ezzel természetesen a többi objektumorientált alapfogalom is más megközelítésbe kerül. A refaktorálás, mint a kód minőségjavítás módszere, foglalkozik a döntésstruktúrák osztály-alosztály kapcsolatokba történő kiemelésének lehetőségeivel; ugyanakkor nem részletezi a döntésredundanciák esetei szerinti döntésösszevonások lehetőségeit, melyet dolgozatom egyik eredményeként a döntésstruktúrák refaktorálással történő optimalizálásának fő lehetőségének gondolok.
A programokban lévő döntésismétlődések megszüntetésére a döntések osztályhierarchiákba történő kiemelésével vagy már kiemelt döntések összevonásával van lehetőség, amely után a döntések lehetőségeihez tartozó adattartalmak és funkcionalitások egy osztály alosztályaiban kapnak keretet. Tehát a döntésösszevonás lehetőségét megadó döntéskiemelés során a döntés definíciót egy absztraktabb formában, azaz egy osztályhierarchiában adjuk meg, ahol a döntés „felületét” egy polimorf metódus jelenti. Kiemelve a döntéseket és a döntéslehetőségeket egy osztályba és annak alosztályaiba, a megfelelő döntésösszevonásokat elvégezve lehetővé válik, hogy a döntéseket egy helyen meghozva, annak eredményét a kiemeléskor, összevonáskor megadott osztályba és annak alosztályaiba „zárva” (példányosítás) archiváljuk. A létrejövő objektumot egy referencia változóval hivatkozzuk, amely típusa a szülőosztály. Az adott döntés további döntési pontjain már a meghozott, archivált döntés eredményét – szülőosztály típusú referencia típusú változót használhatjuk fel úgy, hogy nem kell figyelni arra, hogy a döntés során mely döntéslehetőség érvényesült, azaz melyik alosztályt példányosítottuk, hiszen a szülőosztály típusával hivatkozva a későbbi felhasználások során rejtve, bezárva marad, mivel a döntés felületét egy polimorf metódus adja.
A döntéseknek két állapotát különböztetjük meg attól függően, hogy az adott döntés definiálása, azaz a döntéslehetőségek adatstruktúrái, metodológiája egy metódusban van-e megadva vagy már kiemelten egy osztályban és annak alosztályaiban.

Meghatározom azokat a döntésszerkezeti jellemzőket, amelyek esetén a döntésredundanciák megszűntetése indokolja a döntésösszevonásokat. A döntésösszevonások megfogalmazott esetei kulcsfontosságúak a döntésszerkezetek alapján javasolt tervezési alapelvek és metrikák, tervezési minták jellemzői szempontjából.
Megfogalmazok két új tervezési alapelvet. Az első alapelv – „Öröklődést döntésredundanciák megszűntetésére használjunk!” – egyértelműen meghatározza azon eseteket, amikor az öröklődés, mint objektumorientált eszköz használata az objektumösszetétellel szemben indokolt lehet. Mivel a már meglévő tervezési alapelvek egy kritikus pontja annak eldöntése, hogy az öröklődés használata mely esetekben indokolt, ezért az új elv lényeges kiegészítést jelent a már meglévő alapelvek tekintetében. A döntésredundanciák csökkentése, mint új cél elérése érdekében megfogalmazott másik új alapelv: „A döntésredundanciákat kerüljük!”.
Az elképzelésnek megfelelően az objektumorientált alapeszközök és a már meglévő tervezési alapelvek értelmezését is kiegészítem, amely alapján a tervezési alapelvek céljaival összhangban levő tervezési minták céljai, leírásai is változnak.
Olyan új objektumorientált metrikákat vezetek be, melyekkel lehetőség van a programban szereplő döntésredundanciák mértékének megállapítására, illetve, melyekkel a bevezetett tervezési alapelvek teljesülése mérhető. („Döntésabsztrakciós metrika”, „Döntéskiemeléssel létrejött öröklődések aránya”, „Ekvivalens döntésesetek aránya”, „Döntéspredikátumában eltérő egyéb viselkedésében egyező döntésesetek aránya”, „Ekvivalens döntéspredikátumú döntésesetek aránya” metrikák)
A tervezési minták új értelmezés szerinti célja a jól strukturáltság elérése a döntésismétlődések megszűntetésével. Tehát a tervezési minták a döntéskiemelésekre adnak recepteket, vagyis arra, hogy a különböző strukturált döntési helyzetekben a döntésismétlődés megszüntetése miként biztosítható. A tervezési minták osztályozhatók aszerint, hogy milyen döntésszerkezetek esetén adják meg az optimális döntésdeklarációkat, illetve, hogy a döntésarchiválás mely módját támogatják.
A tervezési minták alkalmazásának egy új irányát vehetjük észre, miszerint a döntések nem kiemelt állapota esetén is észlelhetők azok a döntéskapcsolatok, amelyek az adott tervezési mintára jellemzőek, így a tervezési minták nem csak a jelenlegi formájukban ismerhetők fel, hanem a döntések nem kiemelt állapota esetén a tervezési minták „lenyomatai” is észlelhetők.
Ez új lehetőséget ad annak tisztázására, hogy milyen esetekben lehetséges egy adott tervezési minta felhasználása, hiszen egy tervezési minta leírásakor nem egy optimális megoldás paramétereit kell, hogy megadjuk, hanem egy olyan tervezési helyzet körülményeit, amikor egy adott tervezési minta felhasználásával egy optimálisabb struktúra kialakítása érhető el. Tehát a tervezési minták bevezetésének lehetősége sokkal érthetőbbé válik, mivel konkrét megoldástól független jellemzők, a döntésszerkezet szerint határozható meg egy tervezési minta alkalmazhatósága.
Abban az esetben, ha a döntések elemzése automatizált eszközökkel is megtehető, akkor lehetőség nyílhat a tervezési minta alkalmazás szükségességének automatikus felismerésére.

Hogy az elképzelésnek megfelelően a strukturáltságot, azaz a jól strukturált program irányelveinek teljesülését elemezni tudjuk, szükséges egy olyan formalizációs eszköz, amellyel a döntések viselkedése vizsgálható. Erre a Java programok viselkedés interfész specifikációs nyelvét, a JML-t használom, amely lehetőséget ad a döntés lehetőségeinek adatstruktúráját, metodológiáját logikai kifejezésekre alapozva specifikálni. Az így formalizált döntések vizsgálata a döntésösszevonás szabályai szerint már lehetséges.
A JML alapú döntésspecifikációra alapozva a tervezési mintákat jellemző döntésszerkezet is modellezhető. A „Híd" tervezési mintában található döntésszerkezet deklarációk különböző állapotainak formális leírását külön-külön adom meg, amely alapján megfogalmazom azokat az általános szabályokat, amelyek a JML-es döntésszerkezet szerinti tervezési minta formalizálás alapjai lehetnek.

Az elgondolás helyességének vizsgálataként az elmúlt másfél évben folyó „eFilter" kutatás-fejlesztéses projekt projektvezetőjeként és vezető tervezőjeként a tervezési fázisában kezdeményeztem, hogy a tervezési minták alkalmazhatóságát a döntésszerkezetek elemzésével végezzük, mely során számos komplex tervezési problémára sikerült optimális megoldást találni. A projekt során a döntésszerkezetek – tervezési minták összefüggéseinek egy új irányát sikerült felfedezni, miszerint a tervezési mintákra jellemző döntési helyzetek már a használati eset modellekben is megjelennek. Új kutatási irány ennek megfelelően az, hogy a tervezési minták a használati eset modellekben miként jelennek meg, illetve hogy a tervezési minták döntésszerkezetei hogyan tükröződnek a használati esetek szintjén.

Döntés, döntéskiemelés fogalmainak bevezetése:
Bevezetem a megvalósítástól független döntés fogalmát, amely kifejti, hogy az öröklődéses osztályszerkezetek a döntések absztraktabb deklarációs módjai, illetve kijelenti, hogy a döntéskiemelés során a döntések viselkedése és adatszerkezete nem változik. Ezek a definíciók a refaktorálás témakörében már alkalmazott feltételes utasítások öröklődéssel és polimorf metódussal történő definiálására épülnek.
(3.1 Döntés és döntéskiemelés)

Döntésredundanciák megszűntetése döntésösszevonásokkal:
A döntés, döntéskiemelés fogalmára alapozva bevezetem a döntésösszevonás fogalmát, mint a döntésredundanciák megszűntetésének eszközét. A különböző döntésredundancia típusokra külön adom meg a döntésösszevonás módszerét.
Ezek az esetek a későbbiek folyamán kritikusak, mivel ezek motiválják a tervezési alapelvekkel kapcsolatos javaslatok megszületését, illetve a kód minőség mérés lehetőségeit kiterjesztő metrikák bevezetését. A tervezési minták osztályozása szintén ezen esetekre épül.
(3.2 A döntésredundanciák elkerülésének esetei, 3.3 Döntésösszevonás)

Annak érdekében, hogy azon helyzetek felderíthetők legyenek, amikor a döntéskiemelés, döntésredundanciák feloldása indokolt, szükséges egy formalizációs eszköz. Felvázoltam a program formalizációs eszközök céljait, ismertettem fejlődési irányukat, mellyel a Floyd féle induktív állítások módszerére

és Hoare féle deduktív levezetési rendszerére alapozva kialakult a JML, ami egy Java viselkedés specifikációs eszköz.

Döntéskiemelés során a döntésszerkezetek JML specifikációja miként változik:
A döntések formális leírására és elemzésére a JML-t használom, mely szerint ismertetem az egyszintű és többszintű döntések nem kiemelt és kiemelt állapot szerinti JML specifikációjának tulajdonságait.
A döntésszerkezetek JML alapú formalizálásával kapcsolatban új eredménynek tartom az egyszintű és többszintű döntésszerkezetek JML alapú formalizáció módjának ismertetése során a nem kiemelt és az osztályszerkezetbe kiemelt döntések JML specifikációs lehetőségeinek bemutatását, amely alapján érzékelhető, hogy milyen változások következnek be a viselkedési szerződésben a kiemelés során.
4.2 Döntések formalizálása nem kiemelt állapotban
4.3 Döntések formalizálása kiemelt állapotban

Döntésösszevonás eseteinek JML alapú megfogalmazásai:
A „3.3 Döntésösszevonás" fejezetben ismertetettem azon esetek, amikor a döntésredundancia megszűntetése indokolja a döntésösszevonást. Az ott megfogalmazott kritériumokat JML specifikációs eszközökkel is definiálom a „4.4 Döntésösszevonás eseteinek vizsgálata JML-lel" fejezetben,

„Öröklést döntésredundancia megszűntetésre alkalmazzunk!", „Döntés-redundanciát kerüljük!" tervezési alapelvek bevezetése:
Látnunk kell, hogy a döntésösszevonás esetei a „2.2 Az objektumorientált tervezés alapelvei" fejezetben taglalt számos elvet új megvilágításba helyeznek. A leglényegesebb, hogy a döntésösszevonás esetei egyértelmű segítséget adnak annak eldöntésében, hogy mikor szükséges és indokolt öröklődés, azaz osztályhierarchia bevezetése.
Bevezetek két olyan új objektumorientált tervezési alapelvet, melyek a döntésredundancia elkerülés szabályaiból táplálkozva fogalmaznak meg két új szabályt. Az „Öröklést döntés redundancia megszűntetésre alkalmazzunk!" javasolt alapelv egyértelműen próbálja deklarálni azon eseteket, amikor az öröklődés használata indokolt.
„A döntésredundanciát kerüljük" tervezési alapelv általánosan fogalmazza meg a döntésredundanciák elkerülésének szükségességét, amely a feltételezésem szerint a programminőség javulását eredményezi.
(3.6 Objektumorientált tervezési alapelvek kiegészítése)

Döntésredundanciák mérése új metrikákkal:
Olyan új objektumorientált metrikákat vezetek be, melyekkel lehetőség van a programban szereplő döntésredundanciák mértékének megállapítására.

Ezek közül a legegyszerűbben mérhető a „Döntésabsztrakciós metrika", amely a kiemelt – nem kiemelt (polimorf metódushívások száma – polimorf metódushívások száma+„if-else" utasítások száma) döntésesetek számosságának arányát vizsgálja.

Kísérleti eredményekkel igazolom, hogy a kiemelt döntésesetek arányának növekedése javítja a kód stabilitását.

A „Döntéskiemeléssel létrejött öröklődések aránya” metrikát az „Öröklést döntésredundanciák megszűntetésre alkalmazzunk!” új tervezési alapelv teljesülésének mérésére vezetem be.
Feltételezem, hogy, ha egy osztály-alosztály kapcsolatban van polimorf metódus (a kiemelt döntések felületeit a polimorf metódusok adják), akkor az adott öröklődés döntésredundanciák megszűntetésére jött létre.
Kísérleti eredményekkel igazolom, hogy a döntéskiemeléssel létrejött öröklődések arányának növekedése javítja a kód stabilitását.

A „Ekvivalens döntésesetek aránya”, „Döntéspredikátumában eltérő egyéb viselkedésében egyező döntésesetek aránya”, „Ekvivalens döntéspredikátumú döntésesetek aránya” metrikákkal a döntésredundanciák eseteinek teljesülése mérhető. Ezek felhasználásával szofisztikáltabb képet kaphatunk a döntésszerkezetek állapotáról, melyekhez viszont a döntések viselkedési szerződésének elemzés szükséges. Ezen metrikák alkalmasak „A döntésredundanciát kerüljük!” tervezési alapelv teljesülésének mérésére.
(3.7 Döntésszerkezet elemzése új metrikákkal)

Tervezési minták jellemzése döntésszerkezetekkel:
A tervezési minták jellemzését a tervezési helyzeteket leginkább kifejező döntés szerkezetekkel fejezem ki. A tervezési minták felhasználhatósága így könnyebbé válik, mivel jellemzésük tervezési megoldás független módon, de kellő egzaktsággal tehető meg.
A „5.1 Tervezési minták döntésszerkezetek szerinti értelmezése” alfejezetben a „GOF”könyvben (Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.; Design Patterns: Elements of Reusable Object-Oriented Software; Addison-Wesley Professional Computing Series; 1995.) összegyűjtött tervezési minták döntésszerkezeteit mutatom be.
Az elképzelés alkalmazhatóságának igazolása céljából az „eFilter” kutatásfejlesztési projekt tervezési szakaszában a döntési struktúra elemzésével sikerült felfedezni néhány tervezési minta alkalmazhatóságát, melyek közül egy esetet az „5.2 Sablonfüggvény tervezési minta döntésszerkezet szerinti felismerése az eFilter projekten” fejezetben, mint esettanulmányt ismertetek.

Tervezési minta felismerés, automatikus felismerés lehetősége döntésszerkezet alapján, megvalósítástól függetlenül:
A tervezési minták döntésszerkezete alapján lehetőség van arra, hogy a tervezési minták alkalmazhatóságának lehetőségét a programtervezők könnyebben észrevegyék.
A döntésszerkezet szerinti tervezési minta felismerés esettanulmánya az „5.3 Tervezési minták jellemzése döntéskiemelés előtt és után” fejezetben található, ahol a „*Híd*” tervezési minta döntésszerkezetének megjelenését ismertetem

döntéskiemelés és összevonás előtt és után. Egy példaprogrammal teszem érthetőbbé a „*Híd*” tervezési minta döntésszerkezeteinek megvalósulásait.
A tervezési minták döntésszerkezeteire alapozva a tervezési minták automatikus felismerése is támogatható úgy, hogy a megvalósítástól független döntésszerkezetek szerint tudjuk észrevenni a tervezési minták „lenyomatait” a tervben vagy forráskódban. Ez a tervezési minta automatikus felismerés egy újszerű megközelítése lehet, amit új kutatási lehetőségnek tartok.

Tervezési minták osztályozása döntésszerkezetek szerint:
A tervezési minták osztályozhatók aszerint, hogy milyen döntésszerkezet esetén adják meg az optimális döntésdeklarációkat, mellyel a döntésredundanciák elkerülése biztosítható.
A „5.1.4 Tervezési minták osztályozásának új módszere” fejezetben bemutatott osztályozást olyan új eredménynek tartom, ami hozzájárulhat ahhoz, hogy a tervezési minták döntésszerkezetből fakadó esetleges hasonlóságait észrevehessük, felhasználhatóságukat javítsuk.

Tervezési minták döntésszerkezeteinek JML alapú specifikációja:
A JML lehetőséget ad arra, hogy a tervezési minták döntésszerkezeteit egzakt módon, a döntések reprezentációjának módjától függetlenül deklaráljuk. Így megoldható a döntések viselkedési szerződéseinek összehasonlításával azon döntésszerkezetek felismerése, melyek az egyes mintákra jellemzők.
A tervezési minták JML alapú formalizálását a „*Híd*” tervezési minta döntésszerkezetének JML alapú specifikációjával szemléltetem: „5.4.1 „Híd” tervezési minta JML alapú formális leírása”, amit egy esettanulmánnyal teszek érthetőbbé az „5.4.2 „Híd” tervezési minta JML alapú formális leírása példán keresztül” fejezetben. Ezekre alapozva az „5.4.3 Tervezési minták döntésszerkezetek szerinti JML alapú formalizálásának szabályai” fejezetben összegzem tapasztalataimat.

8 Summary

As I noticed, the most important goal of design patterns and design principles is the elimination of the decision structure based redundancies as a kinds of implementation dependencies. The not properly implemented, redundant decisions result in implementation dependencies, which are called decision structure implementation dependencies.

During the decisions in the application, we decide about the functionality and/or data structure needed at a given point. When we define a decision, we give the functionality and/or data structure of the decision's decision options. By decision option, we mean one possible outcome of a decision, which means the enforcing of the functionality and/or data structure appropriate for the decision option, and the decision predicate which decides which decision option will set off at a given decision location.

In order to understand the options of dissolving the decision redundancies, we need to extend the basic concept of object-oriented methodology, the inheritance: The inheritance is the tool to dissolve the decision redundancies in the software, which gives a more abstract framework to declare decisions after their contraction and raising. With this, all the other basic concepts of object-oriented programming change as well.

Refactoring, as the tool of improving the quality of the code, handles the options of raising the decision structures into class-subclass relations, but in the same time, it does not detail the options of decision contraction according to the cases of decision redundancies, that, as a result of my thesis, I think to be the main option of optimizing the decision structures by refactoring.

To extinguish the decision-recurrence in softwares, we have the option to organize the decisions into class hierarchies or to contract the already organized decisions, after which the data that belongs to the decisions' options and functionalities will get a frame in a class' subclasses. So during the decision-raising that gives the opportunity of the decision-contraction, the decision definition is given in a more abstract form, namely in a class hierarchy, where the "surface" of the decision is meant by a polymorph method. By raising the decisions and decision options into a class and it's subclasses, than executing the appropriate decision contractions, it becomes possible to make the decision in one place and to archive it by closing the results into the same class and it's subclasses given during the raising and contraction (instantiating). The object created this way is referenced with a reference variable which's type is the parent class. On the further decision locations of the given decision we can use the result of archived decision – as a variable with a reference type and parent class type – in a way that we do not need to check what decision option was enforced during the decision, namely which subclass had been instantiated, because it remains hidden and closed during further usage as it is referenced with it's parent class because the surface of the decision is given by a polymorph method.

We differentiate two states of the decisions depending on that if defining the given decision, namely the data structures, methodology of the decision options are given in one method, or are raised to a class and it's subclasses.

I appoint those decision structure parameters, when the elimination of the decision redundancies justifies the decision-contractions. The defined cases of decision-contractions are the keys to the parameters of the design patterns, the designing principles and metrics suggested by the decision structures. metrics.
I formulate two new design principles. The first principle – "Inheritance is used dissolve decision redundancies." – clearly identifies those cases, when the usage of inheritance as object-oriented tool is reasonable over against object composition. As one of the critical points of the existing design principles is the decision of when is the use of inheritance justified, the new principle means a significant supplement in point of the extant principles. In favour of the diminishing of the decision redundancies, as an achievable new goal, the other new principle is the following: "Avoid decision redundancies."
According to the concept, I also expand the interpretation of the basic object-oriented paradigms and the existing design principles, under which the goals and descriptions of the design patterns in consort with the goals of the design principles will change to.
I initiate such new object-oriented metrics, that gives opportunity to determine the degree of decision-redundancies in the software, or rather, with which the fulfillment of the initiated design principles is measurable. (The "Ratio of equivalent decision cases", „Ratio of decision cases different in their decision predicate but the same in any other behavior", „Ratio of decision cases with equivalent decision predicate" "Ratio of inheritances coming into existence by decision-raising", "Metrics of Decision-abstraction" metrics.)
The goal of design patterns according to the new presentation is to achieve a well-structured program by dissolving decision recurrences. So the design patterns give a recipe for decision-raising, or rather to that how can the decision recurrences be avoided in different structured decision situations. The design patterns are classifiable by in what cases of decision structures do they give the optimal decision-declarations, or rather what kind of decision archiving method do they support.
We can notice a new way in using the design patterns, that the decision-structures that are typical to the given design pattern can be noticed even if the decision is not in a raised state, hence the design patterns are not only recognizable in their current form, but the "imprints" of the designing patterns can be noticed in the non-raised state of the decisions.
This gives a new opportunity to clarify in what cases is the usage of a given design pattern is possible, as when describing a design pattern we do not need to give the parameters of an optimal solution, but the circumstances of design situation, when by using the given design pattern a more optimal structure can be developed. Thus the possibility of initiating the design patterns become more understandable, as the applicability of a design pattern can be determined by parameters independent of the factual solution, which is according to the decision structure.
In the case when the analysis of the decision can be done by automated tools, it may be possible to automatically realize the need of using a design pattern.
As to be able to analyze, according to the idea, verify the structuredness, namely the fulfillment of the principles of a well-structured software, we need a formalization

tool that can be used to examine the behavior of decisions. For this, I use the behavior interface specification language of the Java applications, the JML, that enables to specify the data structure, methodology of the decisions' options based on logical expressions. The examination of the decisions formalized in this manner is now possible according to the rules of decision-contraction.
Built upon the JML-based decision specification, the decision structure describing the design patterns can be modeled to. I give the formal descriptions of the different states of the decision structure declarations found in the design pattern one by one, by which I formulate those general rules, that can be the basics of the design pattern formalization according to the JML decision structure.

As the examination of the conception's correctness, as project leader and lead designer of the eFilter research-expansion project going on in the last one and a half year, in the design phase I initiated that the design patterns' applicability be analyzed by the decision structures, during which we have managed to find optimal solution for many complex designing problems. During the project, I have managed to discover a new direction in the connection of the decision structures and design patterns, that the design patterns yet appear in the use case models based on decision situations. According to this, the new research direction is to find out how does the design patterns appear in the use case models, or rather how does the decision structures of the designing patterns reflect on the level of use cases.

Initiating the concepts of Decision and Decision-raising
I initiate the realization–independent concept of decision, which unfolds that the inheritance class structures are the more abstract declaration methods of decisions, or rather it states that during decision-raising the behavior and data structure of the decisions do not change. These definitions are based on defining conditional instructions by inheriting and polymorph method used in the topic of refactoring.
(3.1. Decision and decision-raising)

Dissolving decision redundancies by decision-contractions
Based on the concept of decision and decision-raising, I initiate the concept of decision-contraction, as the tool of dissolving decision redundancies. I give the method of decision-contraction one by one for the different decision redundancy types.
These cases are going to be critical later, as these motivate the birth of the suggestions in connection with the design principles, or rather the initiation of the metrics that allow the measurement of the code's quality. The classification of the design patterns is also based on these.
(3.2. The cases of avoiding decision redundancies, 3.3. Decision-contraction)

In order to be able to explore the situations when the decision-raising, the elimination of the decision redundancies are justified, we need to have a formalization tool. I have outlined the goals of the software formalization tools, I have laid their direction of development, which was used develop the JML, which is a Java behavior specifying device based on Floyd's method of inductive statements and Hoare's deductive derivation system.

How does the JML specification of the decision structures during decision-raising:
I use the JML to the formal description and analysis of the decisions, by which I have laid the JML specification's parameters by the single- and multi-level decisions' raised and non-raised states.
In connection with the JML-based formalization of the decision structures, I find it a new goal: the review of the non-raised and class structure-raised decisions' JML specification options during the review of the JML-based formalization of the single- and multi-level decision structures, that can be used to notice what changes will happen in the behavior contract during the raising.
4.2. The formalization of decisions in non-raised state
4.3. The formalization of decisions in raised state

The JML-based composition of the decision-contraction's cases:
In the 3.3. Decision-contraction chapter, I laid those cases when the elimination of decision redundancy justifies the decision-contraction. I also define the criterias mentioned there by the JML specification tools in the 4.4. Analyzing the cases of decision-contraction by JML chapter.

The initiation of the „Using inheritance to dissolve decision-redundancy", „avoid decision-redundancy" design principles
We have to see that the cases of decision-contraction set into a new light many of the principles discussed in chapter 2.2. The principles of object-oriented design. The most important is that the cases of decision-contraction give an obvious help to decide when the inheritance, namely the initiation of the class hierarchy is required and justified.
I initiate two such new object-oriented design principle, that are using the rules of avoiding decision-redundancy to formulate two new rules. The "Inheritance is used to dissolve decision-redundancy" suggested principle obviously tries to declare those cases, when the use of inheritance is justified.
The "Avoid decision-redundancy" design principle formalizes the need for avoiding decision-redundancy in general, that will result in the improvement of the software's quality.
(3.6. Amendment of the object-oriented design principles)

Measuring the decision-redundancies with new metrics
I initiate such object-oriented metrics, that offers the opportunity to measure the degree of decision-redundancies in the software.

Of these, the most easily measurable is the "decision-abstraction metrics", that analyzes the ratio of the raised and non-raised (number of polymorph method callings / (number of polymorph method callings + the number of "if-else" instructions)) decision cases' cardinality. I assume that the improving ratio of the raised decision cases will improve the stability of the code as well.

The „ratio of inheritances coming into existence by decision-raising" metrics is initiated to measure the fulfillment of the new design principle "Inheritance is used to dissolve decision-redundancies".
I assume that, if there is a polymorph method in a class-subclass connection (the surfaces of the raised decisions are given by polymorph methods), than the given inheritance has come into existence to dissolve decision redundancies.

The "Ratio of equivalent decision cases", „Ratio of decision cases different in their decision predicate but the same in any other behavior", „Ratio of decision cases with equivalent decision predicate" metrics can be used to measure the fulfillment of the decision-redundancy cases. By using these, we can get a more sophisticated picture of the state of decision-structures, for which the analysis of the decisions' behavior contract is needed. These metrics are applicable to measure the fulfillment of the "Avoid decision-redundancies" design principle.
(3.7. Analyzing the decision structures with new metrics)

Characterizing design patterns by decision structures
I express the characterization of the design patterns with the decision structures that express the design situations the most. The usage of the design patterns becomes easier this way, as their characterization can be done independently from a design solution, but with enough exactness.
In the 5.1. Interpreting design patterns according to the decision structures sub-chapter, I introduce the decision structures of the design patterns gathered in the "GOF" book (Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.; Design Patterns: Elements of Reusable Object-Oriented Software; Addison-Wesley Professional Computing Series; 1995.).
To confirm the usability of the idea, I managed to discover the usability of a few design patterns by analyzing the decision structure during the design stage of the eFilter research-development project, of which I review one as a case-study in chapter 5.2. Recognizing a "Template method" design pattern by decision structure in the eFilter project.

Design pattern recognition, the option of automatic recognition by decision structure, independent of realization
By the decision structure of the design patterns, there is an opportunity for the software designers to recognize option of the usability of the design patterns more easily.
The case-study of design pattern recognition by decision structure can be found in chapter 5.3. Design patterns before and after decision raising, where I review the appearance of the "Bridge" design pattern's decision structure before and after raising and contraction. I use an example software to make the realization of the "Bridge" design pattern's decision structures more understandable.
Based on the design patterns' decision structures, the design patterns' automatic recognition is eligible by noticing the design patterns' "imprints" in the plan or in the source code according to decision structures independent of realization. This can

be a new approach to the automatic recognition of design pattern, that I think to be a new research opportunity.

The classification of design patterns by decision structures

The design patterns are classifiable by in case of what decision structure do they give the optimal decision declarations, which are used to ensure the avoidance of decision redundancies.

I find the classification in chapter 5.1.4. A new method to classify design patterns such an accomplishment, that adds to the chance of recognizing the incidental similarities of the design patterns that have their roots in the decision structures and improves their usability.

The JML-based specification of the design patterns' decision structures

The JML gives us the opportunity to declare the design patterns' decision structures in an exact way, independent from the mood of the decision's representation. This way we can solve by comparing the decisions' behavior contracts the recognition of those decision structures that are typical to the certain patterns.

I demonstrate the design patterns' JML-based formalization by the JML-based specification of the "bridge" design pattern's decision structure: 5.4.1. JML-based formal description of the "Bridge" design pattern, which is made more understandable by a case-study in chapter 5.4.2. JML-based formal description of the "Bridge" design pattern through examples. Based on these, I summarize my experiences in chapter 5.4.3. Rules of the JML-based formalization of design patterns' decision structure.

9 Irodalomjegyzék

[1] Aggarwal, K.K.; Singh, Y.; Kaur, A.; Malhotra, R.; Empirical Study of Object-Oriented Metrics; Journal Of Object Technology; Oldalszám: 150-173; 2006.

[2] Aggarwal, K.K.; Singh, Y.; Kaur, A.; Malhotra, R.; Software Design Metrices for Object-Oriented Software; Journal Of Object Technology; Oldalszám: 121-138;2006.

[3] Albin-Amiot, H.; Guéhéneuc, Y.; Meta-Modeling Design Patterns: Application to Pattern Detection and Code Analysis; Workshop on Adaptative Object-Models and Metamodeling Techniques; ECOOP (European Conference on Oriented Programming); 2001.

[4] Badri, M.; Badri,L.; Touré,F.; Empirical Analysis of Object-Oriented Design Metrics: Towards a New Metric Using Control Flow Paths and Probabilities; Software Engineering Research Laboratory, Department of Mathematics and Computer Science, University of Quebec at Trois-Rivieres, Québec, Canada; Oldalszám: 123-142; Journal Of Object Technology; 2009.

[5] Booch, G.; Object Oriented Analysis and Design with Applications; Adison-Wesly; 1994.

[6] Booch, G.; Rumbaugh, J.; Jacobson, I.; The Unified Modeling Language User Guide Second Edition; Addison Wesley Professional; 2005.

[7] Brito e Abreu, F., Melo, W.; Evaluating the Impact of Object-Oriented Design on Software Quality; Proceedings of the 3rd International Software Metrics Symposium (METRICS'96), IEEE; Berlin, Germany; Oldalszám: 90-99; 1996.

[8] Burdy, L.; Cheon, Y.; Cok, D. R.; Ernst, M. D.; Kiniry, J. R.; Leavens, G. T.; Leino, K. R. M.; Poll, E.; An overview of JML tools and applications; International Journal on Software Tools for Technology Transfer (STTT); Oldalszám: 212-232; 2004.

[9] Chalin, P.; Kiniry, J. R.; Leavens, G. T.; Poll, E.; Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2; Concordia University, Montr´eal, Qu´ebec, Canada; University College Dublin, Ireland; Iowa State University, Ames, Iowa, USA; Radboud University Nijmegen, Netherland; Lecture Notes in Computer Science; Oldalszám: 342-363; 2006.

[10] Eden, A. H.; Gil, J.; Hirshfel, Y; Yehudai, A.; Towards a Mathematical Foundation For Design Patterns; Computer Science Department and Department of pure mathematics; Tel-Aviv Univerity; Oldalszám:1-20; IBM Research and Technion

[11] Fisher, K.; C. Mitchell, J.; Notes on typed object-oriented programming; Computer Science Dept.; Stanford University; Stanford; Oldalszám: 844-885; 1994.

[12] Floyd, R.W.; Assigning Meaning to Programs; Proceedings of Symposium on Applied Mathematics; Oldalszám: 19-32; 1967.

[13] Fowler, M.; Beck, K.; Brant, J.; Opdyke, W., Roberts, D.; Refactoring: Improving the Design of Existing Code; Addison Wesley; 1999.

[14] Fowler, M.; Inversion of Control Containers and the Dependency Injection pattern; http://martinfowler.com/articles/injection.html; 2004.

[15] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.; Programtervezési minták: Újrahasznosítható elemek objektumközpontú programokhoz; Addison-Wesley Professional Computing Series/Kiskapu; 1995./2004.

[16] Gupta, S.; Metrics for Object Oriented Software Development; http://javaboutique.internet.com/tutorials/codemetrics

[17] Hoare, C. A. R.; An Axiomatic Basis for Computer Programming; Communications of the ACM; The Queen's University of Belfast, Northern Ireland; Oldalszám: 576-583; 1969.

[18] Kerievsky, J; Refactoring to Patterns; Addison Wesley; 2004.

[19] King, G.; Muir, P.; Richards, N.; Bryzak, S.; Yuan, M.; Youngstrom, M.; Bauer, C.; Balunas, J.; Allen, D.; Andersen, M. R.;Bernard, E.; Karlsson, N.; Roth, D.; Drees, M.;Orshalick, J.; Novotny, M.; Seam - Contextual Components; A Framework for Enterprise Java 2.1.1GA

[20] Kusper, G.; Programtervezési Minták Értelmezése Normálformaként; Networkshop konferenciakiadvány; Oldalszám: 8; Miskolc; Hungary; 2006.

[21] Kusper, G.; Radványi, T.; Jegyzet a projekt labor című tárgyhoz; A program is csak egy termék, olyan, mint egy doboz müzli; Főiskolai jegyzet az informatika tanár MA képzéshez; Eszterházy Károly Főiskola, Eger; Lektorálás alatt. Megjelenés: 2011.

[22] Kusper, G.; Kovács, E.; Márien, Sz.; Kusper, K.; Scheffer, I.; Kiss, B.; Kovács, P.; Winkler, E.; Innovatív Megoldások az eFilter projekten; Informatika a felsőoktatásban 2011; 2011.

[23] Kruchten, P.; Rational Unified Process, An Introduction, Third Edition; Addison Wesley; 2003.

[24] Lamport, L.; The temporal logic of Actions; ACM Transactions on Programming Languages and Systems; Oldalszám: 1–52; 1993.

[25] Leavens, G. T.; Baker, A. L.; Ruby, C.; JML: A Notation for Detailed Design; Behavioral Specifications for Businesses and Systems; Fejezet: 12.; Oldalszám: 175-188; 1999.

[26] Leavens, G. T.; Poll, E.; Clifton, C.; Cheon, Y.; Ruby, C.; Cok, D.; Kiniry, J.; JML Reference Manual; 2004.

[27] Leavens, G. T.; Baker, A. L.; Ruby, C.; Preliminary Design of JML: A Behavioral Interface Specification Language for Java; ACM SIGSOFT Software Engineering Notes; 2006

[28] Leavens, G. T.; Cheon Y.; Design By Contract with JML; Dept. of Computer Science; University of Texas at El Paso; Oldalszám: 1-13; 2006.

[29] Liskov, B. H.; WING, J. N. M.; A Behavioral Notion of Subtyping; MIT Laboratory for Computer Science; Carnegie Mellon University; ACM Transactions on Programming Languages and Systems; Oldalszám: 1811-1841; 1994.

[30] Marien, Sz.; Decision based examination of object-oriented programming and Design Patterns; Teaching Mathematics and Computer Science; Oldalszám: 83-109; 2008.

[31] Marien, Sz.; Decision Based Examination of Object-Oriented Methodology Using JML, Annales Mathematicae et Informaticae; Oldalszám: 95-121; 2008.

[32] Márien, Sz.; Kusper, G.; Understanding Design Patterns as Constructive Proofs; Proceedings of ICAI-2004; Oldalszám: 173-182; Eger; 2004.

[33] Márien, Sz.; Kusper, G.; Automated excavation and detection of Design Patterns; Proceedings of CSCS-2004; Szeged; 2004.

[34] Martin, R. C.; Martin, M.; Agile Principles, Patterns and Practices in C#; Prentice-Hall International Editions; 2006.

[35] Meyer, B.; The role of object-oriented metrics; A shorter variant of this article appeared in Computer (IEEE), as part of the Component and Object Technology department; Oldalszám: 123-125; 1998.

[36] Mikkonen, T.; Formalizing Design Patterns; ICSE'98 - IEEE Computer Society Press; Oldalszám: 115-124; 1998.

[37] Nierstrasz, O.; Survey of Object-Oriented Concepts; In Object-Oriented Concepts, Databases and Applications; ACM Press and Addison-Wesley; Oldalszám: 3-21; University of Geneva; 1989.

[38] http://www.oodesign.com/design-principles.html

[39] Piefel, M.; Object Oriented Software Development; Coursework 'Information Engineering'; Lecture Notes; Department of Computing, University of Bradford; 1996/97.

[40] Rambaugh, J.; Jacobson, I.; Booch, G.; The unified modeling language reference manual Second Edition; Addision-Wesley; 2004.

[41] Rosenberg, L.; E. Hyatt, L.; Software Quality Metrics for Object Oriented System Environments; A report of SATC's research on OO metrics; Software Assurance Technology Center as SATC; 1995.

[42] Rumbaugh, J.; M. Blaha; W. Premerlani; F. Eddy; W. Lorensen; Object-Oriented Modeling and Design; Prentice-Hall International Editions; New Jersey; 1991.

[43] Schreiner, W.; The Java Modeling Language (Part 1), (Part 2); Lecture Notes; Research Institute for Symbolic Computation (RISC) Johannes Kepler University, Linz, Austria

[44] Schreiner, W.; From Types to Contracts: Supporting by Light-Weight Specifications the Liskov Substitution Principle; Research Institute for Symbolic Computation (RISC); Johannes Kepler University Linz, Austria; Oldalszám: 1-35; 2010.

[45] Sebesta, R. W.; Concepts of Programming Languages - Seventh Edition; University of Colorado at Colorado Springs; 2006.

[46] Sun Certified Programmer & Developer for Java 5 Study Guide (Exam 310-055)

[47] Taibi, T.; Chek Ling Ngo, D.; Formal Specification of Design Patterns – A Balanced Approach; Journal Of Object Technology; Oldalszám: 127-140; Multimedia University; Malaysia; 2003.

[48] Tar, B.; Some Object-Oriented Design Principles; Lecture Notes; Computer Science and Electrical Engineering Department; University of Maryland Baltimore County; 2004.

[49] Walls, C; Breidenbach, R.; Spring In Action Second Edition; Manning Publications Co.; 2008.