

**Budapesti Műszaki és Gazdaságtudományi Egyetem**

Villamosmérnöki és Informatikai Kar

Irányítástechnika és Informatika Tanszék

Koroknai Máté

A MOF SZABVÁNY IMPLEMENTÁLÁSA

Konzulens

Dr. Simon Balázs

BUDAPEST, 2023

**Tartalomjegyzék**

[**Összefoglaló 4**](#_gjdgxs)

[**Abstract 5**](#_30j0zll)

[**1 Bevezetés 6**](#_1fob9te)

[1.1 Mi az XMI formátum? 6](#_3znysh7)

[1.2 A MOF (Meta-Object Facility) ismertetése 7](#_2et92p0)

[1.3 A MetaDslx keretrendszer bemutatása 8](#_tyjcwt)

[1.4 Tervezés 9](#_3dy6vkm)

[1.5 A dolgozat felépítése 10](#_1t3h5sf)

[**2 Irodalomkutatás 11**](#_4d34og8)

[**3 A MOF generálásához szükséges algoritmusok 12**](#_2s8eyo1)

[3.1 A package merge algoritmus 13](#_17dp8vu)

[3.1.1 Elem átmásolása package merge során 15](#_3rdcrjn)

[3.1.2 Két elem összefésülése package merge során 16](#_26in1rg)

[**4 Irodalomjegyzék 18**](#_lnxbz9)

[**Függelék 19**](#_35nkun2)

**Hallgatói nyilatkozat**

Alulírott **Koroknai Máté**, szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző, cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2023. 05. 20.

...…………………………………………….

Koroknai Máté

**Összefoglaló**

# Abstract

# Bevezetés

Egy modell olyan elemekből áll, amelyek valamilyen valóságos, absztrakt vagy hipotetikus világot írnak le. A modell kialakításához szükséges az absztrakció, ami lehetővé teszi, hogy a felesleges információkat elhanyagoljuk az adott kontextusban. Ezentúl szükséges az osztályozás, ami pedig azt jelenti, hogy a vizsgált dolgokat közös tulajdonságaik és viselkedésük alapján csoportosítsuk.

Metamodellnek nevezzük egy modell modellezési nyelvét. A metamodellezésnek több szintje van, mivel egy metamodell elemeit is le kell tudni írni valahogy. A metamodell adja meg a modell elemeinek struktúráját, szemantikáját és kényszereit. Meta-metamodellnek pedig azokat a modellezési nyelveket hívjuk, amelyekkel metamodelleket lehet alkotni. Ilyen modellezési nyelv a Meta-Object Facility (MOF).

A MOF szabályrendszer implementációja lehetővé teszi, hogy programozhatóan építsünk/kezeljünk egyedi metamodelleket általános felhasználásra (erre példa a Unified Modeling Language, UML), vagy egy adott problémakör megoldásának érdekében. Ez nem csak informatikában hasznos eszköz, hanem más szakterületek körében is (pl. kémiában, biológiában). Rendszerbiológiában már 2004-ben használták a UML egy testreszabott változatát biológiai folyamatok és rendszerek modellezésére.

A MOF implementációjához hasonló eszközök már léteznek, amelyek közül az egyik Eclipse-ben használható: az Eclipse Modeling Framework, amely főleg MOF 1.x kompatibilis, és támogatja Essential MOF (EMOF) metamodellek importálását az XML Metadata Interchange (XMI) formátum által. Ennek a metamodellnek a magja az Ecore, amely nagyjából megyegyezik az EMOF-fal (Essential MOF), de nem teljesen.

## Modellvezérelt fejlesztés és szakterület specifikus nyelvek

A modellvezérelt architektúra (model-driven architecture, MDA) a szoftverfejlesztés egy olyan megközelítése, amely platformfüggetlen iránymutatásokat ad a specifikációk leírásához. A specifikációk így modellként vannak kifejezve. Az architektúra szétválasztja az üzleti- és alkalmazás logikát a platformtól, tehát a modell gyakorlatilag bármilyen platformon megvalósíthatóvá válik, és a szoftver üzleti- és technológiai aspektusai külön-külön tudnak alakulni, fejlődni. Az üzleti logika az új üzleti igényekhez alkalmazkodik, míg a technológia kiaknázhatja az új fejlesztéseket. Az architektúrát az Object Management Group (OMG) definiálta a 2000-es évek elején.

A platformfüggetlen modellek a Meta-Object Facility (MOF) alapján készült modellezési nyelvek által építhetők fel. Ezek a modellek időtállóak és stabilak maradnak, ahogy a technológia fejlődik, maximalizálva a tervezésbe fektetett erőfeszítések megtérülését. Az MDA fejlesztőeszközei segítségével a platformfüggetlen modell platformspecifikus modellé alakítható, majd elkészíthető az implementációja szinte bármilyen köztesszoftver platformra (pl. Web Services, EJB, C# .Net, stb.). A hordozhatóság és az interoperabilitás bele van építve az architektúrába. A modellvezérelt fejlesztést többek között a MOF, a Unified Modeling Language (UML) és az XML Metadata Interchange (XMI) támogatják. Ezekről bővebben a következő szekciók szólnak. (<https://www.omg.org/mda/specs.htm>)

## Mi az XMI formátum?

Az XMI az XML Metadata Interchange rövidítése. Jelenlegi verziója a 2.5.1, és az OMG párhuzamosan fejleszti a UML és a MOF standardokkal. A nevéből eredően arra szolgál, hogy könnyen, konzisztensen lehessen modellek és objektumok leírását megosztani egymással, vagy számítógépekkel.

A következő aspektusokat definiálja objektumok leírására:

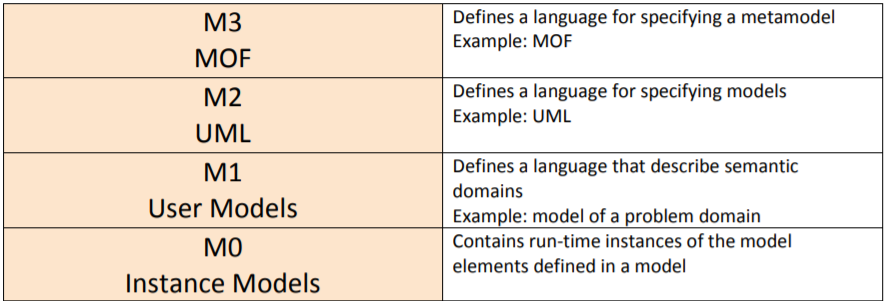
* Objektumok reprezentációja XML elemek és attribútumok segítségével
* Standard mechanizmusok objektumok összelinkeléséhez ugyanazon a fájlon belül, vagy fájlok között
* XMI dokumentumok validálása XML sémák alapján
* Objektum azonosság, amely segítségével hivatkozni lehet objektumokra más objektumokból ID-k és UUID-k alapján

Bármilyen metaadat leírására alkalmas, ami megfelel a MOF szabványnak. Leggyakoribb használata a UML modellek megosztása, viszont alkalmazható mindenféle metamodell szerializálására. Mivel ez egy XML alapú formátum, emberi olvasásra viszonylag alkalmatlan, ezért léteznek olyan eszközök, amelyekkel grafikus, vagy más, felhasználó barátabb módon, lehet megjeleníteni a benne tárolt adatokat. Az egyik ilyen például a jelen diplomamunka tárgya, amely segítségével be lehet olvasni egy XMI fájl tartalmát, és metamodell fájl generálható belőle, amely ember számára lényegesen emészthetőbb.

## A MOF (Meta-Object Facility) ismertetése

A MOF 2.5 az OMG által fejlesztett metamodellezési standard. A UML 2 leegyszerűsített osztály modellezési képességeire van alapozva. Metamodellek definícióján kívül a modellek kezeléséhez is lényeges eszközöket nyújt, például azonosítók (**Identifiers**), általános címkézés (**generic Tag**), reflektív műveletek (**Reflective operations**), amelyek generikusan vannak definiálva, és használhatóak a metamodelltől függetlenül.

A MOF egy nyílt és platform független metaadat (adat az adatról) kezelési keretrendszert biztosít a modell-, vagy metaadat alapú rendszertervezéshez. A metamodell hierarchia az alábbi táblázatban látható.

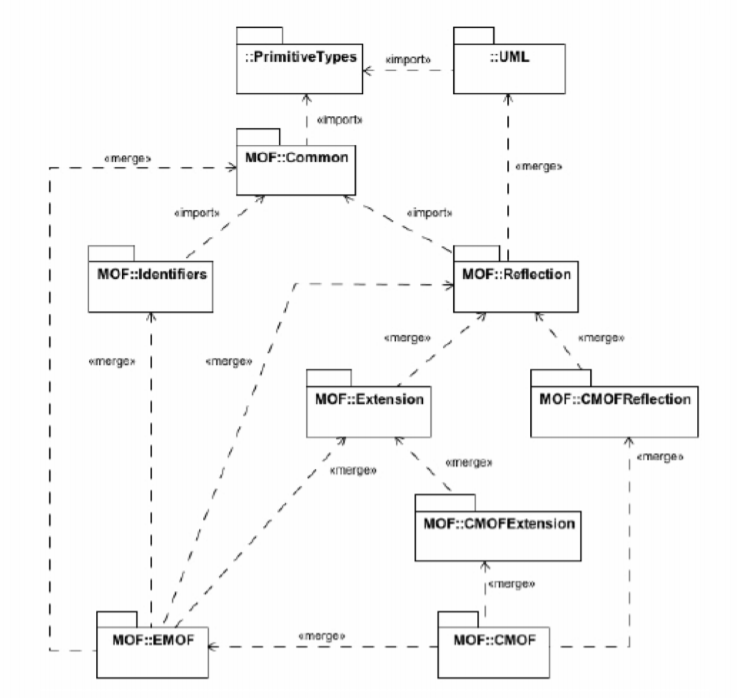


**táblázat 1: Modell hierarchia**

A hierarchia a MOF-fal kezdődik a legfelső absztrakciós szinten, és a lejjebb lévő szintek a fölöttük lévőktől függnek. Felülről a második szinten találhatók a modellezési nyelvek – például a UML – amelyekkel modellek alkothatók. Ezek a metamodellek. Az eggyel lejjebbi szinten helyezkednek el azok a modellek, amelyek a metamodellezési nyelvek használatával születnek, ilyen például egy UML osztálydiagram. A legalsó szint pedig a modellek példányaiból, azaz a modellek valamilyen programozási nyelven vagy keretrendszerben történő megvalósításaiból áll.

Mivel a MOF egy metamodell definíciós eszköz, és maga is egy metamodell, így saját maga által definiálható, és saját maga által is van definiálva.

Az OMG MOF specifikációjának két fő része van: az Essential MOF (EMOF) és a Complete MOF (CMOF). A teljes szabályrendszer – ahogy a nevéből is kiderül – a CMOF, amely tartalmazza az EMOF-ot, és mindkettő a UML 2.5-ből van származtatva. Ezekről a kapcsolatokról a következő diagram árulkodik, amelyen látható, hogy a UML egy részhalmaza belekerül az EMOF-ba és a CMOF-ba is, a MOF::Reflection csomag által.



**ábra 1: A MOF felépítése [1]**

Jelen diplomamunka elsődleges célja az egyes ábrán látható package merge-ök megvalósítása.

## A MetaDslx keretrendszer bemutatása

A MetaDslx egy felhasználóbarát, a tanszék által fejlesztett metamodellezési keretrendszer a .NET Standard 2.0-ra építve. Az alábbiakra nyújt megoldást:

* DSL-ek (domain specific language) implementása
* Zár nélküli szálbiztos immutable semantic object modellek
* Automatikusan generált szintaktikai és szemantikai fordítás DSL-ekről objektum modellekre
* Szemantikai annotációkkal kiterjesztett ANTLR4 nyelvtanra alapozott, Roslyn stílusú publikus API-val rendelkező fordítók
* Felhasználóbarát „template” alapú kód generátorok
* Szintaxis kiemelők Visual Studiohoz
* Az OMG metamodell standardok teljes implementációja: UML, MOF
* Modellek szerializációja és deszerializációja az XMI standardon keresztül
* Az Eclipse EMF ecore metamodell implementációja

Ezek közül jelen diplomamunka feladata a MOF implementációja, amely később a keretrendszer részét fogja képezni.

A MetaDslx keretrendszer három egyedi nyelvet biztosít DSL-ek létrehozására. Ezek a Meta-Model nyelv (\*.mm), a Meta-Generator nyelv (\*.mgen) és a Meta-Compiler nyelv (\*.ag4). A Meta-Model nyelvvel emberi olvasásra alkalmas módon lehet metamodelleket leírni. A keretrendszer ebből automatikusan generál C# osztályokat. Ezekre a C# osztályokra épül a metamodell implementációja. Az implementáció tartalmazza a szálbiztos immutábilis osztályokat, és a szálbiztos builder osztályokat.

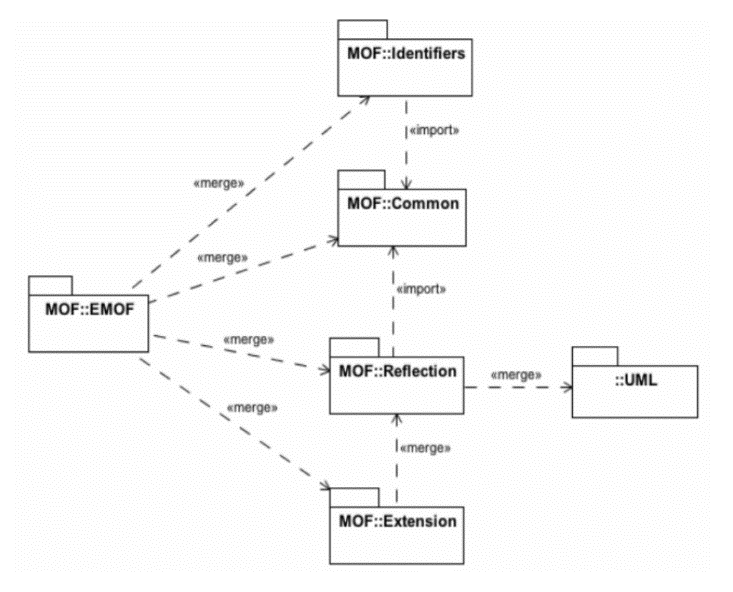
A Meta-Generator nyelv egy felhasználóbarát template nyelv, amely programkód, vagy bármilyen más kimenet generálására szolgál. Amikor a generátort használjuk, egy C# osztály van végrehajtva, amit a keretrendszer a \*.mgen forráskódból generált. Ez az osztály végzi el az összes műveletet, ami a generátorban volt definiálva. A generátor bármilyen .NET objektum gráfon végig tud haladni, beleértve egy metamodell példányt.

A Meta-Compiler egy ANTLR4 nyelvtan, annotációkkal kiegészítve, amelyek szemantikai információt tartalmaznak. Ilyen nyelvtanokkal új programozási nyelveket lehet definiálni. A Meta-Compiler kód alapján a keretrendszer Roslyn stílusú immutable fordítót generál, amely a gyors ANTLR4-et használja elemzéshez. A szemantikai annotációk mentén szemantikai analízist végez, és automatikusan csinál egy példányt a metamodellből.

A Meta-Model-, a Meta-Compiler-, és a Meta-Generátor nyelvek együttes használatával szálbiztos, zármentes, immutábilis objektumgráfokon alapuló nagy teljesítményű doménspecifikus nyelvek építhetők.

## Tervezés

A MOF implementálása két „bootstrap” projekt segítségével történik. Az egyikben egy minimális MOF modell van összerakva kézzel, amely képes a MOF szabvány XMI általi betöltésére és manipulálására olyan szinten, amennyire a teljes modell generálásához szükség van. A másik projekten belül három osztályban vannak implementálva az Essential MOF és a Complete MOF felépítéséhez szükséges algoritmusok. Ezeknek az osztálydiagramja a … ábrán látható.

A MergeHelper statikus osztály általánosan használható függvényekből áll, amelyek a „package merge” szabályai alapján működnek, míg a UmlToMofHelper osztályban konkrétan a UML magjának beolvasztására szolgáló függvények találhatók. A MofGenerator osztály pedig az imént említett két segédosztály felhasználásával generálja a kész MOF modellt. Ennek az osztálynak a 

A Bootstrap módon történő generálásra az ad lehetőséget, hogy az Essential MOF a package merge segítségével van definiálva. Alapvetően az lenne ideális, ha az EMOF csak kiterjesztené a UML modellt alosztályok által, amelyek újabb tulajdonságokat és metódusokat vezetnek be. Ezután az EMOF formálisan meghatározható lenne az EMOF által, anélkül, hogy package merge-re szükség lenne. Ez azonban nem elegendő, mivel a MOF::Reflection csomagban be kell vezetni az Object ősosztályt, amely az osztály hierarchia tetejére kerül. Ettől fog örökölni a UML::Element osztály is, és ezért szükséges a package merge folyamata. Ennek eredménye az, hogy az EMOF egy különálló modell, amely beolvasztja a UML-t, viszont nem örököl tőle. Annak köszönhetően, hogy az EMOF a package merge által van definiálva, azt is biztosítjuk, hogy az EMOF frissítve lesz, ha a UML-t bármilyen módosítások érik.

**ábra : EMOF modell CITATION Obj19 \l 1033 [1]**

A két segédosztályon kívül hibakeresési célokkal bevezetésre kerül egy harmadik segédosztály is.

## A dolgozat felépítése

Az ismertetőket követő irodalomkutatással foglalkozó rövid fejezet említ hasonló projekteket, a 3. fejezet pedig a MOF generálásához szükséges algoritmusokat mutatja be. Ezután a 4. fejezet a kész implementáció tesztelését tárgyalja, először egy példa modell által, később pedig az egység tesztek bemutatásával.

# Irodalomkutatás

Ahogy a bevezetőben szó esett róla, hasonló implementációk már léteznek különböző környezetekben, viszont nem sok munka található a témában. Ezek főleg a MOF 1.x-szel kompatibilisek, és egyik sem egyezik meg teljesen a MOF-fal.

Az egyik ilyen implementáció a korábban említett Ecore, amely az Eclipse Modeling Framework (EMF) metamodell magját képezi [2]. Az EMF eszközöket ad arra, hogy futásidőben lehessen Java osztályokat és adapter osztályokat generálni a modell specifikációja alapján (XMI formátumból). Lehetővé teszi a modell megtekintését, és parancs alapú szerkesztését is egy egyszerű szerkesztőben. EMF három fő részből áll:

* EMF: azon túl, hogy az Ecore metamodellt tartalmazza, futásidejű támogatást ad modellekhez, XMI szerializáláshoz és ad egy hatékony, reflektív API-t a modellek általános manipulációjához
* EMF.Edit:
* EMF.Codegen:

A Xactiumtól származó XMF-Mosaic az ExMOF-ot (Executable MOF) implemetálja, amely a részhalmaza és a kiterjesztése a MOF 1.x-nek [2].

Létezik a Sun által fejlesztett UML2MOF eszköz, amely UML metamodelleket transzformál MOF 1.x metamodellekké. Szintén a Sun-tól származik az MDR implementáció is [2].

Kereskedelmi implementációk készültek az Adaptive és a Compuware látal [2].

Mivel ezek zárt forráskódú, vagy nem pontos implementációi a MOF-nak, ezért a diplomamunka alapjául kizárólag az OMG által kiadott szöveges és XMI szabványok szolgálnak.

# A MOF generálásához szükséges algoritmusok

Előkészítő lépésként a MOF modellt generáló programnak be kell töltenie a MOF- és a UML specifikációt, amely XMI formátumban elérhető az OMG weboldalán. Ezután mivel a MOF több csomagból áll össze, ezekre a csomagokra külön-külön hivatkozik a program, hogy később könnyebb legyen dolgozni velük. Ezek a csomagok a MOF::Reflection, MOF::Extension, CMOF::Reflection, MOF::Common, MOF::Identifiers, MOF::EMOF és a MOF::CMOF, ahogy az 1-es ábrán látható.

Ezt követően a UML-nek azon részét kell összefésülni (merge-ölni) a Reflection csomaggal, amelyek a MOF szöveges specifikációjában szerepelnek. A további lépések:

* Reflection csomag merge-ölése az Extension csomagba
* Common csomag merge-ölése az EMOF-ba
* Identifiers csomag merge-ölése az EMOF-ba
* Reflection csomag merge-ölése az EMOF-ba
* Extension csomag merge-ölése az EMOF-ba
* Ha CMOF generálására van szükség, akkor további lépések kellenek:
  + Extension merge-ölése a CMOFExtension csomagba
  + Reflection merge-ölése a CMOFReflection-be
  + CMOFExtension merge-ölése a CMOF-ba
  + EMOF csomag merge-ölése a CMOF csomagba
* Olyan ősosztályok hozzáadása az EMOF vagy CMOF modellhez, amelyektől a UML-ből bemerge-ölt osztályok örökölnek
* Olyan metatípusok hozzáadása a UML-ből az EMOF vagy CMOF modellhez, amelyek megjelennek a UML bemerge-ölése után

Ezeket követően az EMOF vagy a CMOF modell generálása történik a keretrendszer segítségével:

var generator = new MofModelToMetaModelGenerator(cmof.ToImmutable().PackagedElement);

var generatedCode = generator.Generate("MofImplLib.Model", "Mof", "url");

File.WriteAllText("MofImplLib/Model/Mof.mm", generatedCode);

A generált modell a Mof.mm fájlban lesz, amiből a keretrendszer automatikusan generálja a C# forráskódot is. A következő lépés a modellben lévő osztályok metódusainak implementálása, ami már kézzel történik. Ezeknek nagy része ugyanaz, mint a UML modellben, tehát át lehet őket másolni.

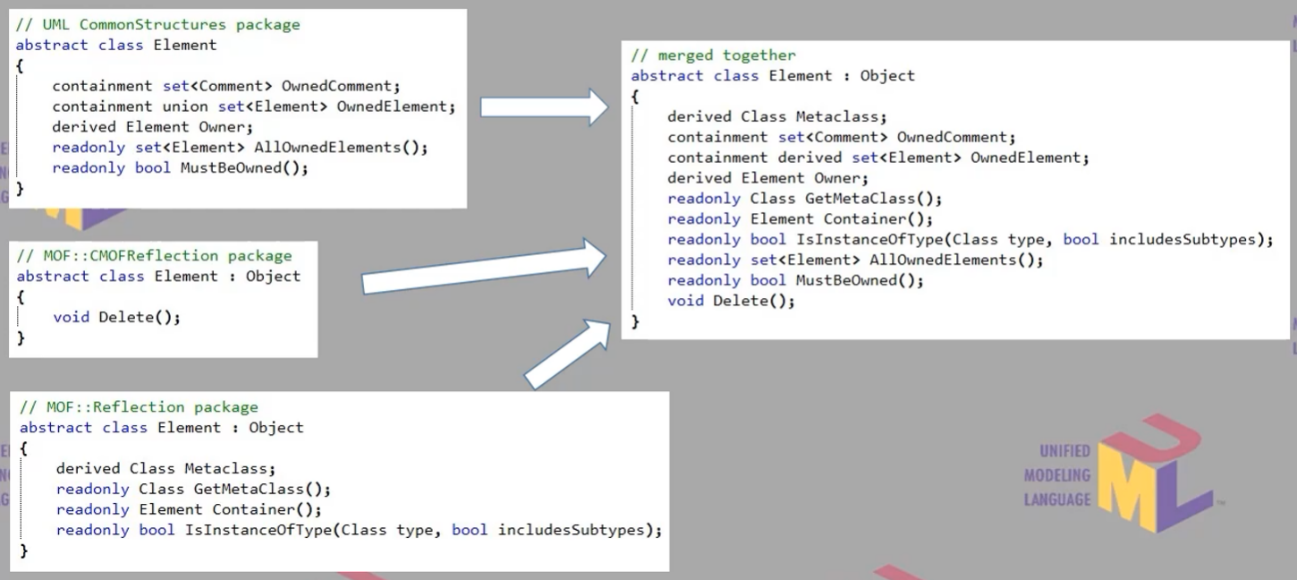
## A package merge algoritmus

Erről a UML szöveges specifikációjának 12.2.3.2-es fejezete tartalmazza a szabályokat. A terminológia:

* *merged package (forrás csomag)*: az a csomag, amelyet belefésüljük a fogadó csomagba.
* *receiving package (fogadó csomag)*: az a csomag, amelybe belekerülnek a forrás csomag elemei, és az összefésülés eredményét fogja tartalmazni, viszont ez az elnevezés még az összefésülés előtti állapotra vonatkozik.
* *resulting package (eredmény csomag)*: az a csomag, amely már tartalmazza a package merge folyamat eredményét. Ez ugyanaz a csomag, mint a fogadó csomag, de már a folyamat utáni állapotáról van szó.
* *merged element (forrás elem)*: olyan elem, amelyet a forrás csomag tartalmaz.
* *receiving element (fogadó elem)*: olyan elem, amely a fogadó csomagban van. Ha van ehhez az elemhez passzoló forrás elem, akkor a kettő kombinációjából születik az eredmény elem. Ez az elnevezés még az összefésülés előtti állapotra vonatkozik.
* *resulting element (eredmény elem)*: az eredmény csomag egy eleme a package merge folyamat végrehajtása után. Ha egy forrás elemhez nem volt passzoló fogadó elem, akkor az összefésülés után az eredmény elem megegyezik a forrás elemmel. Ha egy fogadó elemhez nem volt passzoló forrás elem, akkor az eredmény elem megegyezik a fogadó elemmel.
* *element type (elem típus)*: bármilyen típusú TypedElement-nek a típusa. Például egy Parameter-nek, vagy egy StructuralFeature-nek a típusa.
* *element metatype (elem metatípus)*: egy modell elemnek a MOF típusa (például Classifier, Association, Feature).

Két csomag összefésülése során az egyik a forrás csomag, a másik pedig a fogadó csomag. Ekkor a forrás csomag nem változik, és a fogadó csomag kiegészítésével születik az eredmény csomag, amely tartalmazza a fogadó összes eddigi elemét/funkcióját és a forrás csomag összes elemét/funkcióját is.

Ha a forrás csomag olyan elemeket tartalmaz, amelyek a nevük alapján nincsenek benne a fogadó csomagban, akkor egyszerűen át kell másolni őket a fogadó csomagba. Ha névütközés van, akkor a két elemet össze kell fésülni. Tehát például osztályok esetében érdekes az Element, mert benne van a UML-ben, a MOF::Reflection-ben és a MOF::CMOFReflection-ben is. A fésülési folyamat eredménye az alábbi ábrán látható:



**ábra 3: Element osztályok összefésülése**

A MOF.xmi nem tartalmaz információt arról, hogy a UML-nek pontosan melyik részét kell merge-ölni, ezért a MOF szöveges specifikációja alapján kézzel kell felvenni azokat a metaosztályokat, amelyekre szükség lesz. Ez a lista az EMOF esetében a szöveges specifikáció 12.4-es -, míg a CMOF esetében a 14.4-es részében található.

Az EMOF-ban felhasznált UML metaosztályok listája:

public static readonly string[] UML\_CLASSES\_TO\_BE\_MERGED\_INTO\_EMOF =

{

// based on Mof2.5.1 specification, section 12.4

"Association",

"Class",

"Comment",

"DataType",

"Enumeration",

"EnumerationLiteral",

"Generalization",

"InstanceSpecification",

"InstanceValue",

"LiteralBoolean",

"LiteralInteger",

"LiteralNull",

"LiteralReal",

"LiteralString",

"LiteralUnlimitedNatural",

"Operation",

"Package",

"Parameter",

"PrimitiveType",

"Property",

"Slot"

};

### Elem átmásolása package merge során

A MOF-ban előforduló elemek: primitív típusok (integer, real, string, stb.), enumerációk, osztályok és asszociációk. Mikor egyik modellből a másikba másolunk elemet, először kérni kell egy azonos típusú üres példányt a factory-tól, ami a modellbe generál elemeket:

MofFactory fctry = new MofFactory(mofModel, ModelFactoryFlags.DontMakeObjectsCreated);

ClassBuilder element = fctry.Class();

Ezután az elem összes tulajdonságát le kell másolni hasonló módon. Tehát ha például egy osztály egy metódusának a másolására van szükség, akkor:

OperationBuilder newOp = fctry.Operation();

newOp.Name = op.Name;

newOp.Visibility = op.Visibility;

newOp.IsReadOnly = op.IsReadOnly;

newOp.IsAbstract = op.IsAbstract;

newOp.IsQuery = op.IsQuery;

...

És így tovább. Egészen addig kell a factory-val generálni üres elemeket, ameddig olyan szintre nem jutunk, ahol már nem kell klónozni (például primitív típusok). Egy osztály metódusának a paramétereit is a factory segítségével másoljuk le:

ParameterBuilder newParam = fctry.Parameter();

newParam.Name = param.Name;

newParam.Direction = param.Direction;

newParam.IsOrdered = param.IsOrdered;

newParam.IsStream = param.IsStream;

newParam.IsUnique = param.IsUnique;

newParam.Visibility = param.Visibility;

newParam.Type = param.Type;

...

newOp.OwnedParameter.Add(newParam);

...

element.OwnedOperation.Add(newOp);

### Két elem összefésülése package merge során

A MOF csomagok összefésülése folyamán csak osztályok összefésülésére van szükség, mivel más elemeknél (például enumeration) nincs ütközés. Az erre vonatkozó szabályokat a UML szöveges specifikációja tartalmazza a 12.2.3.3-as szekcióban. Ezek közül a fontosabbak:

* Olyan forrás-, vagy fogadó elem, amelyre nincs illeszkedő merge-ölendő elem, azt bele kell mély-másolni (deep copy) az eredmény csomagba. Ez azt jelenti, hogy azokat az elemeket is belemásoljuk az eredmény csomagba, amelyekre az elem hivatkozik.
* Ha a forrás- és a fogadó elem teljesen megegyezik, akkor az összefésülés eredménye a fogadó elem.
* Ha van a forrás elemhez passzoló fogadó elem, akkor a metatípusukhoz tartozó transzformációs szabályok alapján lesznek egyesítve.
* Minden olyan típussal rendelkező elemre való típushivatkozás, amely az eredmény csomagba kerül, a megfelelő eredmény TypedElement-re történő hivatkozásra lesz átalakítva (vagyis nem annak inkrementális megfelelőjére).
* Ha mindkét összefésülendő elemnek privát láthatósága van, akkor az eredmény elem is privát lesz. Egyébként pedig publikus.
* Classifier elemek összefésülése esetében, ha mindkét elemre isAbstract = true, akkor az eredmény elem is isAbstract = true. Egyéb esetekben pedig isAbstract = false.
* Classifier elemek összefésülésénél, ha mindkét elemre isFinalSpecialization = true, akkor az eredmény elem is isFinalSpecialization = true. Egyébként az eredmény elem: isFinalSpecialization = false.

# Tesztelés

## Egy tesztmodell

Az kész MOF implementáció segítségével egy olyan metamodell készült, amellyel egyszerű gráfokat lehet modellezni. Két osztályt tartalmaz: Vertex és UndirectedGraph. A Vertex osztály az alábbi kód segítségével generálható:

MutableModel model = new MutableModel("TestGraph");

MofFactory factory = new MofFactory(model);

PrimitiveTypeBuilder stringType = factory.PrimitiveType();

PrimitiveTypeBuilder intType = factory.PrimitiveType();

stringType.Name = "String";

intType.Name = "Integer";

ClassBuilder vertex = factory.Class();

vertex.Name = "Vertex";

// vertex id property

PropertyBuilder vertexId = factory.Property();

vertexId.Type = stringType;

vertexId.Name = "ID";

// NeighborCount attribute

PropertyBuilder vertexNeighborCount = factory.Property();

vertexNeighborCount.Type = intType;

vertexNeighborCount.Name = "NeighborCount";

// VertexNeighbors attribute

PropertyBuilder vertexNeighbors = factory.Property();

var vertexNeighborsUpperValue = factory.LiteralUnlimitedNatural();

vertexNeighborsUpperValue.Value = long.MaxValue;

vertexNeighbors.Type = vertex;

vertexNeighbors.Name = "Neighbors";

vertexNeighbors.IsOrdered = true;

vertexNeighbors.IsUnique = true;

vertexNeighbors.UpperValue = vertexNeighborsUpperValue;

// adding attributes to vertex

vertex.OwnedAttribute.Add(vertexId);

vertex.OwnedAttribute.Add(vertexNeighborCount);

vertex.OwnedAttribute.Add(vertexNeighbors);

// AddEdge method

OperationBuilder vertexAddEdge = factory.Operation();

vertexAddEdge.Name = "AddEdge";

ParameterBuilder vertexAddEdgeInput = factory.Parameter();

vertexAddEdgeInput.Name = "neighbor";

vertexAddEdgeInput.Direction = ParameterDirectionKind.In;

vertexAddEdgeInput.Type = vertex;

vertexAddEdge.OwnedParameter.Add(vertexAddEdgeInput);

// RemoveEdge method

OperationBuilder vertexRemoveEdge = factory.Operation();

vertexRemoveEdge.Name = "RemoveEdge";

ParameterBuilder vertexRemoveEdgeInput = factory.Parameter();

vertexRemoveEdgeInput.Name = "neighbor";

vertexRemoveEdgeInput.Direction = ParameterDirectionKind.In;

vertexRemoveEdgeInput.Type = vertex;

vertexRemoveEdge.OwnedParameter.Add(vertexRemoveEdgeInput);

// Adding methods to vertex

vertex.OwnedOperation.Add(vertexAddEdge);

vertex.OwnedOperation.Add(vertexRemoveEdge);

A generált metamodell kód így néz ki:

class Vertex

{

string ID;

int NeighborCount;

list<Vertex> Neighbors;

void AddEdge(Vertex neighbor);

void RemoveEdge(Vertex neighbor);

}

És az ebből generált C# API az alábbi:

public interface Vertex : global::MetaDslx.Modeling.ImmutableObject

{

string ID { get; }

int NeighborCount { get; }

global::MetaDslx.Modeling.ImmutableModelList<Vertex> Neighbors { get; }

void AddEdge(Vertex neighbor);

void RemoveEdge(Vertex neighbor);

new VertexBuilder ToMutable();

new VertexBuilder ToMutable(global::MetaDslx.Modeling.MutableModel model);

}

public interface VertexBuilder : global::MetaDslx.Modeling.MutableObject

{

string ID { get; set; }

void SetIDLazy(global::System.Func<string> lazy);

void SetIDLazy(global::System.Func<VertexBuilder, string> lazy);

void SetIDLazy(global::System.Func<Vertex, string> immutableLazy,

global::System.Func<VertexBuilder, string> mutableLazy);

int NeighborCount { get; }

void SetNeighborCountLazy(global::System.Func<int> lazy);

void SetNeighborCountLazy(global::System.Func<VertexBuilder, int> lazy);

void SetNeighborCountLazy(global::System.Func<Vertex, int> immutableLazy,

global::System.Func<VertexBuilder, int> mutableLazy);

global::MetaDslx.Modeling.MutableModelList<VertexBuilder> Neighbors { get; }

void AddEdge(VertexBuilder neighbor);

void RemoveEdge(VertexBuilder neighbor);

new Vertex ToImmutable();

new Vertex ToImmutable(global::MetaDslx.Modeling.ImmutableModel model);

}

Innen már csak az interfészben definiált metódusokat szükséges implementálni, hogy a

## Unit tesztek

Unit tesztelésre a MofImplementation osztályban implementált metódusoknak van szüksége, arra az esetre, ha frissítenek a MOF illetve a UML metamodellen. A tesztelés az MSTest keretrendszer segítségével történik Visual Studio-ban.

A teszteléshez szükséges konstansokat két segédosztály: a MofTestConstantsGen és a TestInstances generálja. A MofTestConstantsGen betölti a UML metamodellt, amelynek a segítségével létrehoz egy példányt a TestInstances osztályból. A TestInstances osztály tölti be a UML modellt és kikeres belőle egy-egy példányt mindegyik MOF osztályból, amelyhez tartozik tesztelendő metódus. A UML modell azonban nem tartalmaz elemet mindegyik MOF osztályból. Ilyen esetekben a TestInstances osztály készít egy példányt a hiányzó osztályból, és hozzáadja a betöltött UML modellhez.

Ezután a MofTestConstantsGen osztály mindegyik tesztelendő példányon meghívja a szükséges metódust és a kimenetét elmenti egy C# fájlban az alábbi formátumban:

static class TestConstants

{

public static readonly string[] Association\_EndType =

{

"ObjectNode",

"Behavior",

}; // (A\_selection\_objectNode)

public static readonly string[] BehavioralFeature\_InputParameters =

{

"redefiningElement",

}; // (isConsistentWith)

...

}

# Irodalomjegyzék

|  |  |
| --- | --- |
| [1] | O. M. Group, „OMG Meta Object Facility (MOF) Core Specification,” 2019. |
| [2] | R. Paige, „The Meta-Object Facility (MOF),” University of York, UK, 2006. |
| [3] | O. M. Group, „OMG Unified Modeling Language,” OMG, 2017. |

**Függelék**