Desenvolupament de programari dirigit per models

Francisco Durán Muñoz
Javier Troya Castilla
Antonio Vallecillo Moreno

PID_00184448

UOC
Universitat Oberta de Catalunya

Índex

Introducció

El desenvolupament de programari dirigit per models (denominat *MDD* pel seu acrònim en anglès, *model-driven development*) és una proposta per al desenvolupament de programari en la qual s'atribueix als models el paper principal, enfront de les propostes tradicionals basades en llenguatges de programació i plataformes d'objectes i components de programari.

El propòsit d'MDD és tractar de reduir els costos i temps de desenvolupament de les aplicacions de programari i millorar la qualitat dels sistemes que es construeixen, amb independència de la plataforma en què serà executat el programari i garantint les inversions empresarials enfront de la ràpida evolució de la tecnologia.

Els pilars bàsics sobre els quals es recolza MDD són els models, les transformacions entre models i els llenguatges específics de domini. Aquests són precisament els temes que es cobreixen en aquest mòdul. També discutirem els avantatges que ofereix MDD, i també els principals reptes i riscos que implica adoptar-lo actualment.

Objectius

Aquest mòdul introdueix els principals conceptes, mecanismes i processos relacionats amb el desenvolupament de programari dirigit per models, i se centra sobretot en una proposta concreta com és la *model-driven architecture* (MDA®) de l'OMG™ (Object Management Group).

Més concretament, els objectius que persegueix aquest mòdul són els següents:

1. Donar a conèixer el concepte de *desenvolupament de programari dirigit per models*, juntament amb els seus principals i mecanismes característiques. També es presenten els avantatges i inconvenients que planteja usar-lo per al disseny i desenvolupament de sistemes de programari.

2. Introduir la metamodelització com una tècnica essencial dins d'MDA per a definir llenguatges de modelització específics de domini.

3. Presentar els llenguatges que ofereix l'OMG per al desenvolupament de llenguatges específics de domini, i en particular OCL i les extensions d'UML mitjançant perfils UML.

4. Introduir els conceptes i mecanismes relacionats amb les transformacions de models, i en particular els que ofereix el llenguatge de transformació ATL.

5. Presentar els principals estàndards relacionats amb el desenvolupament de programari dirigit per models.

1. Desenvolupament dirigit per models (MDA i MDE)

1.1. Introducció

El desenvolupament de programari dirigit per models sorgeix com a resposta als principals problemes que tenen actualment les companyies de desenvolupament de programari: d'una banda, per a gestionar la creixent complexitat dels sistemes que construeixen i mantenen, i per l'altra, per a adaptar-se a la ràpida evolució de les tecnologies de programari.

En termes generals, són diversos els factors que fan d'MDD, i en particular d'MDA (*model-driven architecture*), la proposta adequada per a abordar aquests problemes per part de qualsevol companyia que vulgui ser competitiva en el sector del desenvolupament de programari.

En primer lloc, s'usen **models** per a representar tant els sistemes com els artefactes de programari mateixos. Cada model tracta un aspecte del sistema, que pot ser especificat a un nivell més elevat d'abstracció i de manera independent de la tecnologia utilitzada. Aquest fet permet que l'evolució de les plataformes tecnològiques sigui independent del sistema mateix, i així en disminuïm les dependències.

En segon lloc, s'aconsegueix també protegir gran part de la inversió que es fa en la informatització d'un sistema, ja que els models són els veritables artífexs del seu funcionament final i, per tant, no serà necessari començar des de zero cada vegada que es plantegi un nou projecte o es vulgui fer algun tipus de manteniment sobre el producte.

En tercer lloc, en MDA els models constitueixen la documentació mateixa del sistema, se'n disminueix considerablement el cost associat i se n'augmenta la mantenibilitat, ja que la implementació es fa de manera automatitzada a partir de la documentació mateixa.

Aquest capítol presenta els principals conceptes que intervenen en el desenvolupament de programari dirigit per models, les notacions utilitzades i les tècniques i eines més comunes que s'utilitzen en aquest àmbit.

1.2. Conceptes bàsics

1.2.1. Abstracció

A l'hora de dissenyar i desenvolupar qualsevol aplicació de programari cal disposar de llenguatges que permetin representar tant l'estructura com el comportament dels sistemes de manera precisa, concreta, correcta i adequada. Per a això és fonamental introduir tres conceptes clau: ***abstracció***, ***expressivitat*** i ***complexitat***.

Abstreure significa destacar una sèrie de característiques essencials d'un sistema o objecte, des d'un punt de vista determinat, ignorant aquelles altres característiques que no són rellevants des d'aquesta perspectiva.

Consulta recomanada

E. W. Dijkstra (1972). "The Humble Programmer". A: *ACM Turing Lecture*. <http://www.cs.utexas.edu/~EWD/transcriptions/EWD03xx/EWD340.html>

En paraules d'Edsger W. Dijkstra,

> "Ser abstracte no significa en absolut ser imprecís [...]. El propòsit principal de l'abstracció és definir un nivell semàntic en el qual poder ser totalment precís."

Per això és vital determinar el punt de vista des del qual cal plantejar el problema, que identifica de manera precisa tant els conceptes que s'han de tenir en compte a l'hora de plantejar el problema i la seva solució, com aquells aspectes que podem ignorar perquè no són rellevants des d'aquest punt de vista.

Lectura recomanada

És interessant l'anàlisi que es fa d'aquest tema en l'article de Jean Bezivin (2005) "The Unification Power of Models", *SoSym* (vol. 4, núm. 2, pàg. 171-188).

Pensem, per exemple, en els diferents llenguatges i notacions que s'utilitzen a l'hora de descriure una ciutat. Així, tenim el plànol del metro, que mostra les parades i les connexions entre aquestes, però ignora les distàncies entre les estacions, perquè són irrellevants en aquest nivell de detall. D'altra banda, hi ha la llista de carrers, que mostra els carrers i les distàncies relatives entre aquests, però ignora altres dades, com per exemple l'orografia de la ciutat, és a dir, les inclinacions dels carrers i els desnivells (cosa important si estem interessats a planificar una carrera urbana, per exemple). Per a això hi ha el plànol topogràfic, que mostra els nivells d'altitud sense fixar-se en els carrers concrets o en els edificis. És possible també disposar avui dia, per exemple, de mapes de soroll d'una ciutat, amb el grau de contaminació acústica a diferents hores del dia i de la nit. Cadascun d'aquests plànols usa una notació diferent, la més adequada per a descriure els conceptes que es pretenen expressar, i cadascun se centra en uns aspectes essencials, i n'ignora uns altres. Per descomptat, seria possible tenir un plànol amb absolutament tota la informació, però, igual que la realitat, seria massa complex per a entendre'l, manejar-lo i raonar-hi.

1.2.2. Expressivitat

Un altre dels factors importants per considerar és el de l'expressivitat del llenguatge que usem (sigui de programació, de modelització, etc.) per a dissenyar el sistema, descriure els requisits, implementar la solució, etc.

L'**expressivitat** és la capacitat d'un llenguatge de poder descriure determinats elements i les característiques d'un sistema de manera adequada, completa, concisa i sense ambigüitat.

Sempre és molt important tenir el millor llenguatge per a descriure el que volem expressar. És per això que actualment hi ha nombrosos llenguatges de programació, cadascun indicat per a expressar determinats aspectes:

- declaratius i funcionals per a especificar els tipus de dades i els algorismes,
- orientats a objectes per a descriure comportament de sistemes de programari reactius,
- llenguatges de consulta per a accedir a bases de dades,
- de programació web per a desenvolupar aplicacions sobre Internet, etc.

Això és similar al que ocorre en altres disciplines d'enginyeria, en què es disposa de llenguatges visuals per a dissenyar els plànols d'un edifici, fórmules matemàtiques per a calcular les càrregues de les estructures, textos per a descriure els requisits de seguretat, etc.

1.2.3. Complexitat

La complexitat és un altre dels conceptes clau que s'han de tenir en compte a l'hora de dissenyar i desenvolupar qualsevol sistema de programari. Una de les definicions de complexitat més simples i fàcils d'entendre diu:

Un sistema és **complex** quan no és possible que una sola persona sigui capaç de comprendre'l, manejar-lo i raonar-hi.

Fred Brooks, en el seu famós article "No Silver Bullet", distingeix entre dos tipus de complexitat: essencial i accidental.

Consulta recomanada

F. Brooks (1986). "No Silver Bullet: Essence and Accidents in Software Engineering". A: *Proceedings of the IFIP 10th World Computing Conference* (pàg. 1069-1076).

També a *Computer* (vol. 20, núm. 4 (1987, abril), pàg. 10-19), i cap. 17 de *Mythical Man-Month, Silver Bullet Refired*, Addison Wesley, 1995.

La **complexitat essencial** d'un problema o d'un sistema és la que es deriva de la seva naturalesa mateixa, la que impedeix l'existència de solucions "simples".

Les aplicacions per a gestionar un aeroport o una central nuclear són intrínsecament complexes, ja que ho són els sistemes que s'han de gestionar.

La **complexitat accidental**, no obstant això, deriva de l'adequació dels llenguatges, algorismes i eines que s'utilitzen per a plantejar la solució a un problema.

Vegeu també

Vegeu el que s'ha estudiat en l'assignatura *Disseny d'estructures de dades* del grau d'Enginyeria en Informàtica.

En general, cada tipus de llenguatge de programació ofereix una sèrie de mecanismes que el fan més (o menys) apropiat al tipus de problema que tractem de resoldre i al tipus de sistema que pretenem desenvolupar.

Vegeu també

Estudiarem amb detall la reutilització en el mòdul "Desenvolupament de programari basat en reutilització".

Collar un cargol es converteix en una tasca complexa si solament tenim un martell. De la mateixa manera, dissenyar aplicacions web es converteix en una tasca molt difícil si tractem de fer-les usant llenguatges com l'assemblador o el COBOL.

Consulta recomanada

C. Szyperski (2002). *Component Software - Beyond Object-Oriented Programming* (2a. ed.). Addison Wesley.

És important assenyalar que l'expressivitat d'un llenguatge influeix directament en la complexitat accidental de la solució, encara que no és l'únic factor que s'ha de tenir en compte.

Si el que volem és ordenar un vector de nombres podem utilitzar diferents algorismes, cadascun més apropiat depenent de la naturalesa i l'estat previ de les dades (encara que tots estiguin expressats amb el mateix llenguatge de programació). Així, el mètode d'ordenació per inserció és el més adequat si les dades estan ja gairebé ordenades prèviament, i *quicksort* és el més eficient si estan molt desordenades.

1.2.4. Reutilització

A més de l'abstracció com a mecanisme per a abordar la complexitat, un altre dels objectius de qualsevol disciplina d'enginyeria és el de la reutilització. En comptes de repetir l'especificació d'un codi, una estructura o un conjunt de funcions, la reutilització els permet definir una sola vegada i utilitzar-los diverses, encara que, com molt encertadament diu Clemens Szyperski:

"Reutilitzar no significa solament usar més d'una vegada, sinó poder usar en diferents contextos."

Citació

Un programari *framework* és una aplicació parametritzable que proporciona un conjunt de biblioteques i serveis per mitjà d'un conjunt d'interfícies. Més informació a: **M. Fayad; D. C. Schmidt** (1997). "Object-Oriented Application Frameworks". *Comms. of the ACM* (vol. 40, núm. 10).

Mecanismes de reutilització

La recerca de mecanismes de reutilització millors ha estat una altra de les constants en l'evolució de la programació. Així, les macros i funcions dels llenguatges de programació van permetre reutilitzar grups de línies de codi, i els tipus abstractes van ser definits per a agrupar diverses funcions entorn de certs tipus de dades. Els objectes permeten encapsular l'estat i comportament dels elements d'una mateixa classe en unitats bàsiques. No obstant això, aviat es va veure que la granularitat dels objectes era insuficient per a abordar grans sistemes distribuïts i van aparèixer els components, que agrupaven un nombre d'objectes relacionats entre si per oferir una sèrie de serveis per mitjà d'interfícies ben definides, amb la propietat de gestionar-ne de manera conjunta el cicle de vida i el desplegament (*deployment*). Basats en components, els marcs de treball (*frameworks*) van representar un avenç important perquè permetien la reutilització no solament d'una sèrie de components, sinó també de la seva arquitectura de programari. Finalment, els serveis web han permès canviar el model d'"instal·la-i-usa" dels components de programari pel d'"invoca-i-usa", i han resolt el problema de l'actualització de versions locals, la particularització de components a plataformes de maquinari o programari concretes, etc.

Encara que els avenços continus en els llenguatges i mecanismes de programació són realment significatius, veiem que encara no s'han resolt tots els problemes, i que els sistemes d'informació que es construeixen continuen sense oferir solucions de reutilització a gran escala. De fet, una crítica molt comuna

entre els màxims responsables de les grans companyies és la seva dependència absoluta de determinades plataformes de maquinari i programari, de llenguatges de programació, de tecnologies concretes de bases de dades, etc. Ells perceben una enorme volatilitat en les tecnologies de maquinari i programari, que evolucionen a gran velocitat i deixen obsolets els seus sistemes d'informació en molt pocs anys –quan encara ni tan sols s'ha amortitzat la inversió feta. I la informàtica, l'única cosa que els ofereix ara mateix és que actualitzin tots els seus sistemes i els substitueixin per altres de nous, i per a això cal començar gairebé des de zero a dissenyar-los i després migrar totes les seves dades i serveis a aquestes noves plataformes.

Un dels principals problemes amb aquestes migracions, a part del cost i temps que representen, és que els tornen a deixar amb un conjunt de programes i dades dependents d'una plataforma concreta, i a una escala d'abstracció que no en permet la reutilització quan d'aquí a uns anys calgui actualitzar-ne de nou les aplicacions (i calgui començar-ne els dissenys gairebé des del principi una altra vegada).

Un altre problema important que tenen aquestes empreses és el de la interoperabilitat amb els sistemes d'informació dels seus clients i proveïdors. L'heterogeneïtat actual representa un impediment important per a l'intercanvi de dades, funcions i serveis entre aplicacions. Una de les principals raons per a aquests problemes d'interoperabilitat és el baix nivell de detall en el qual es descriuen els sistemes, els serveis i les seves interfícies. No és suficient de descriure les operacions proporcionades o l'ordre en el qual cal invocar-les, sinó que també és necessari tenir en compte el model de negoci de totes dues empreses, els seus contextos legals i fiscals, els contractes de qualitat de servei, les responsabilitats per fallades en el servei, etc.

Per això seria convenient descriure els sistemes a un nivell d'abstracció que permetés la reutilització de les dades i processos de negoci, de manera independent dels llenguatges de programació, la tecnologia subjacent i les plataformes en què s'han d'executar els programes, i que a més facilités la interoperabilitat amb altres sistemes externs.

1.2.5. Models

Un dels termes clau en la filosofia de l'MDD és el concepte de ***model***.

De manera senzilla podríem definir un model com una abstracció simplificada d'un sistema o concepte del món real.

No obstant això, aquesta no és l'única definició que trobarem en la bibliografia sobre el terme *model*. A manera d'exemple, les citacions següents mostren algunes de les accepcions més comunes d'aquest concepte en el nostre context:

- Un model és una **descripció** d'un sistema, o part d'aquest, escrit en un llenguatge ben definit (Warmer i Kleppe, 2003).

- Un model és una **representació** d'una part de la funcionalitat, estructura o comportament d'un sistema (OMG, *Model Driven Architecture - A technical perspective*, 2001).

- Un model és una **descripció** o **especificació** d'un sistema i el seu entorn definida per a un cert propòsit (OMG, *MDA Guide*, 2003).

- Un model **captura una vista** d'un sistema físic, amb un cert propòsit. El propòsit determina el que ha de ser inclòs en el model i el que és irrellevant. Per tant, el model descriu aquells aspectes del sistema físic que són rellevants al propòsit del model, i al nivell d'abstracció adequat (OMG, *UML Superstructure*, 2010).

- Un model és un **conjunt de sentències** sobre un sistema (Seidewitz, "What models mean", *IEEE Computer*, 2003). En aquesta referència, Seidewitz entén per *sentència* una expressió booleana sobre el sistema, que pot estar representada tant gràficament com textualment.

- M és un model de S si M pot ser utilitzat per a **respondre preguntes** sobre S (D. T. Ross i M. Minsky, 1960).

Vegeu també

Les referències bibliogràfiques mostrades aquí les podeu trobar en l'apartat de "Bibliografia".

Web recomanat

És interessant també consultar el que defineix la RAE com a model: http://buscon.rae.es/draeI/SrvltConsulta?LEMA=modelo.

Basant-nos en aquestes, podem concloure en la definició següent:

> Un **model** d'un cert <X> és una **especificació** o **descripció** d'aquest <X> des d'un punt de vista determinat, expressat en un llenguatge ben definit i amb un propòsit determinat.

En aquesta definició, <X> representa l'objecte o sistema que volem modelitzar, i que pot ser tant una cosa concreta com una cosa abstracta. De fet, el model proporciona una "especificació" de <X> quan es dóna la circumstància que <X> no existeix encara (per exemple, és el sistema per construir), mentre que proporciona una "descripció" quan representa un sistema que ja existeix en la realitat i que volem modelitzar amb algun propòsit determinat[1].

(1)Per exemple, analitzar-ne alguna de les característiques o propietats, entendre'n el funcionament, abstreure'n algun dels aspectes, etc.

1.2.6. Ús i utilitat dels models

Encara que no hi ha consens en quines són les característiques que han de tenir els models per a ser considerats útils i efectius, una de les descripcions més encertades és la de Bran Selic, un dels fundadors de l'MDD i pioner en l'ús de les seves tècniques. Segons ell, els models haurien de ser:

Consulta recomanada

Vegeu, per exemple, la presentació que Bran Selic va fer al congrés MODPROD a Suècia el 2011: "Abstraction Patterns in Model-Based Engineering".

- **Adequats**: construïts amb un propòsit concret, des d'un punt de vista determinat i dirigits a un conjunt d'usuaris ben definit.

- **Abstractes**: emfatitzen els aspectes importants per al seu propòsit alhora que oculten els aspectes irrellevants.

- **Comprensibles**: expressats en un llenguatge fàcilment entenedor per als usuaris.

- **Precisos**: representen fidelment l'objecte o sistema modelitzat.

- **Predictius**: poden ser usats per a respondre preguntes sobre el model i inferir conclusions correctes.

- **Rendibles**: han de ser més fàcils i barats de construir i estudiar que el sistema mateix.

Bran Selic també assenyala les funcions principals que els models haurien de tenir en l'àmbit de l'enginyeria del programari:

- **Comprendre** el sistema
  - L'estructura, el comportament i qualsevol altra característica rellevant d'un sistema i el seu entorn des d'un punt de vista determinat.
  - Separar adequadament cadascun dels aspectes i descriure'ls al nivell conceptual adequat.

- Servir de **mecanisme de comunicació**
  - Amb els diferents tipus de *stakeholders* del sistema (desenvolupadors, usuaris finals, personal de suport i manteniment, etc.).
  - Amb les altres organitzacions (proveïdors i clients que necessiten comprendre el sistema a l'hora d'interoperar-hi).

Stakeholder

Persona o entitat que està implicada en l'adquisició, desenvolupament, explotació o manteniment d'un sistema de programari.

- **Validar** el sistema i el seu disseny
  - Detectar errors, omissions i anomalies en el disseny tan aviat com sigui possible (com més aviat es detectin menys costa corregir-los).

  - Raonar sobre el sistema, inferint propietats sobre el seu comportament (en cas de models executables que puguin servir com a prototips).

  - Poder fer anàlisis formals sobre el sistema.

- **Guiar** la implementació
  - Servir com a "plànols" per a construir el sistema i que en permetin guiar la implementació d'una manera precisa i sense ambigüitats.

  - Generar, de la manera més automàtica possible, tant el codi final com tots els artefactes necessaris per a implementar, configurar i desplegar el sistema.

Una cosa que diferencia l'enginyeria del programari de la resta de les enginyeries és que en aquestes últimes el medi en el qual es construeixen els plànols i models és molt diferent del mitjà en el qual finalment es construeixen els edificis, ponts o avions. L'avantatge que en l'enginyeria del programari el medi sigui el mateix (els ordinadors) permetrà que es puguin definir transformacions automàtiques que siguin capaces de generar la implementació a partir dels models d'alt nivell, una mica més costós en altres disciplines. Per això en el cas d'MDD (i, en particular en MDA) el que es persegueix és que la generació de les implementacions d'un sistema siguin el més automatitzades possibles, cosa que és factible gràcies a l'ús de les transformacions de models.

Vegeu també

Les transformacions de models s'estudien amb més detall en l'apartat 4.

1.2.7. Metamodels

És important assenyalar el fet que el llenguatge que s'usa per a descriure el model ha d'estar ben definit i oferir un nivell d'abstracció adequat per a expressar el model i per a raonar-hi. En aquest sentit, la idea compartida per tots els paradigmes englobats dins de l'MDD és la conveniència d'utilitzar per a la modelització llenguatges de més nivell d'abstracció que els llenguatges de programació, això és, llenguatges que manegin conceptes més propers al domini del problema, denominats *llenguatges específics de domini* (o DSL, *domain-specific language*). Aquests llenguatges requereixen al seu torn una descripció precisa; encara que aquesta es pot donar de diferents maneres, en el món d'MDD el normal i més apropiat és definir-los com a models també. Com abordarem amb més detall en l'apartat 3, la metamodelització és una estratègia molt utilitzada actualment per a la definició de llenguatges de modelització.

Vegeu també

Els llenguatges específics de domini s'estudien més endavant en l'apartat 3.

Un **metamodel** és un model que especifica els conceptes d'un llenguatge, les relacions entre aquests i les regles estructurals que restringeixen els possibles elements dels models vàlids, i també aquelles combinacions entre elements que respecten les regles semàntiques del domini.

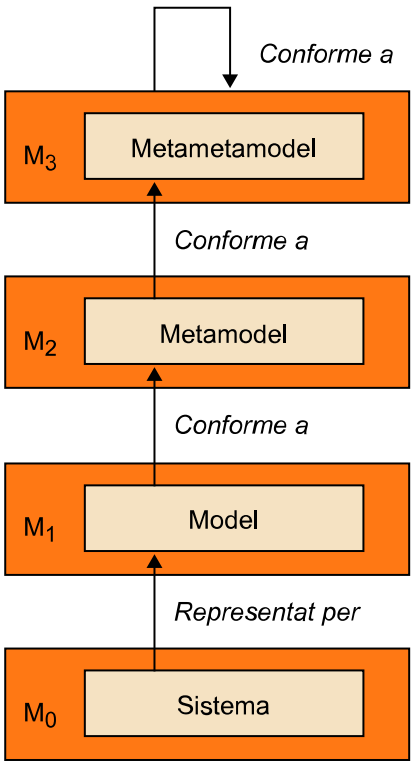
El metamodel d'UML és un model que conté els elements per a descriure models UML, com *package, classifier, class, operation, association*, etc. El metamodel d'UML també defineix les relacions entre aquests conceptes, i també les restriccions d'integritat dels models UML. Un exemple d'aquestes restriccions són les que obliguen al fet que les associacions només puguin connectar *classifiers*, i no paquets o operacions.

D'aquesta manera, cada model s'escriu en el llenguatge que defineix el seu metamodel (el seu llenguatge de modelització), i queda establerta la relació entre el model i el seu metamodel per una relació de "conformitat" (i direm que un model és "conforme a" un metamodel).

Nota

Hi ha gent que sol dir que un model és una "instància" del seu metamodel, però això és similar a dir que un programa és una instància de la seva gramàtica, la qual cosa no és del tot correcte.

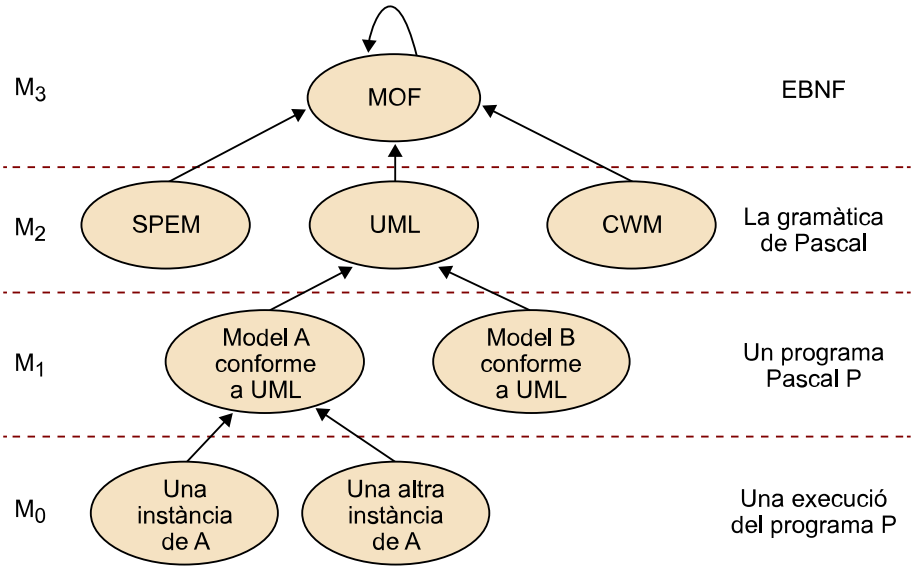
Figura 1. L'organització en quatre nivells de l'OMG



Arribats a aquest punt, no oblidem que els metamodels són al seu torn models i, per tant, estan escrits igualment en el llenguatge definit pel seu metametamodel. Segons la proposta de l'OMG, el procediment recursiu de definir models conformes a altres models de més grau d'abstracció acaba quan s'aconsegueix el nivell de metametamodel, ja que els metametamodels es diu que són conformes a ells mateixos, tal com s'il·lustra en la figura 1.

La figura 2 representa una instanciació concreta de l'arquitectura en quatre nivells representada en la figura 1. En aquesta, MOF (*meta-object facility*) és el llenguatge de l'OMG per a descriure metamodels; i UML, SPEM i CWM (*common warehouse metamodel*) són llenguatges de modelització conformes a MOF (per a descriure sistemes, processos de negoci i magatzems de dades, respectivament). La figura 2 il·lustra també l'analogia existent entre aquesta organització en quatre nivells i la jerarquia de nivells dels llenguatges tradicionals de programació (representada en el lateral dret).

Figura 2. Exemple de l'organització en quatre nivells de l'OMG



Nota

EBNF: Extended Backus-Naur Form, defineix una manera formal de descriure la gramàtica d'un llenguatge de programació.

1.2.8. Transformacions de models en MDA

D'importància particular en MDA és la noció de *transformació entre models*.

Una **transformació de models** és el procés de convertir un model d'un sistema en un altre model del mateix sistema. Així mateix, anomenem així l'especificació d'aquest procés.

En essència, una transformació estableix un **conjunt de regles** que descriuen com un model expressat en un llenguatge origen pot ser transformat en un model en un llenguatge destinació.

Nota

Com veurem en l'apartat 4, les transformacions de models són també models, la qual cosa permet fer un tractament unificat de tots els artefactes que intervenen en qualsevol procés MDD.

Per a definir una transformació entre models és necessari:

1) Seleccionar el tipus de transformació i el llenguatge de definició de transformacions que s'ha d'utilitzar.

2) Seleccionar l'eina que ens permeti implementar els models i les transformacions de manera automàtica.

1.3. Terminologia

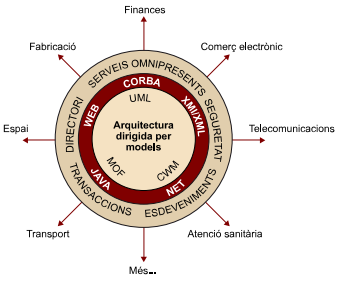
La comunitat utilitza actualment un conjunt d'acrònims referits a diferents enfocaments relacionats amb l'enginyeria del programari que usa models com a elements clau dels seus processos: MDD, MBE, MDA, etc. Encara que alguns ja s'han esmentat en seccions anteriors, en aquest apartat tractarem d'aclarir aquests conceptes i les diferències entre ells.

MDD (*model-driven development*) és un paradigma de desenvolupament de programari que utilitza models per a dissenyar els sistemes a diferents nivells d'abstracció, i seqüències de transformacions de models per a generar uns models a partir d'altres fins a generar el codi final de les aplicacions en les plataformes destinació.

MDD té un enfocament clarament descendent, i augmenta en cada fase el nivell de detall i concreció dels models generats per les transformacions.

En MDD el codi es pot considerar com un model més, encara que alguns autors distingeixen entre models i text, i parlen no solament de transformacions "model a model", sinó també de transformacions "model a text". Per a altres, la representació d'un codi font com un model o amb línies de codi és una qüestió merament sintàctica.

MDA (*model-driven architecture*) és la proposta concreta de l'OMG per a implementar MDD, usant les notacions, mecanismes i eines estàndard definits per aquesta organització.



MDA va ser la primera de les propostes d'MDD, i la que va aconseguir fer arrencar l'ús i adopció dels models com a peces clau del desenvolupament de programari. MDA, igual que MDD, advoca per la separació de l'especificació de la funcionalitat d'un sistema independentment de la seva implementació en qualsevol plataforma tecnològica concreta, i en l'ús de transformacions de models per a transformar uns models en altres fins a obtenir les implementacions finals.

Els estàndards que l'OMG ofereix per fer MDA inclouen, entre d'altres:

- **UML** (*unified modeling language*) com a llenguatge de modelització.
- **MOF** (*meta-object facility*) com a llenguatge de metamodelització.
- **OCL** (*object constraint language*) com a llenguatge de restriccions i consulta de models.
- **QVT** (*query-view-transformation*) com a llenguatge de transformació de models.
- **XMI** (*XML metadata interchange*) com a llenguatge d'intercanvi d'informació.

Vegeu també

L'UML s'ha vist en l'assignatura *Enginyeria del programari* del grau d'Enginyeria Informàtica. La resta d'estàndards es presenten en aquest mòdul.

Nota

Metamodelitzar consisteix a definir llenguatges de modelització.

MDE (*model-driven engineering*) és un paradigma dins de l'enginyeria del programari que advoca per l'ús dels models i les transformacions entre aquestes com a peces clau per a dirigir totes les activitats relacionades amb l'enginyeria del programari.

En aquest sentit, MDE és un terme més ampli que MDD, ja que MDD se centra fonamentalment en les tasques de disseny i desenvolupament d'aplicacions, mentre que MDE abasta també la resta de les activitats d'enginyeria de programari: prototipatge i simulació, anàlisi de prestacions, migració d'aplicacions, reenginyeria de sistemes heretats, interconnexió i interoperabilitat de sistemes d'informació, etc.

En altres paraules, la diferència entre MDD i MDE és la mateixa que hi ha entre desenvolupament i enginyeria.

MBE (*model-based engineering*) és un terme general que descriu els enfocaments dins de l'enginyeria del programari que usen models en algun dels seus processos o activitats.

En general, MBE és un terme més ampli que MDE, i que l'engloba com a paradigma. Encara que la diferència és subtil, i principalment de perspectiva, en MDE són els models i les transformacions entre aquests els principals "motors" que guien els processos de desenvolupament, verificació, reenginyeria, evolució o integració. No obstant això, en MBE els models i les transformacions no necessàriament tenen aquest paper "motriu".

Un procés que usi models per a representar tots els artefactes de programari però l'execució del qual es faci de manera completament manual es podria considerar d'MBE però no d'MDE.

Una altra diferència és que MDE se sol centrar en activitats d'enginyeria del programari, mentre que MBE es pot referir també a enginyeria de sistemes o industrials, quan aquests usen models per a representar els seus artefactes o processos.

BPM (*business process modeling*) és una branca del *model-based engineering* que se centra en la modelització dels processos de negoci d'una empresa o organització, de manera independent de les plataformes i les tecnologies utilitzades.

Hi ha diverses notacions actualment per a expressar models de negoci, i les principals són BPMN 2.0 (*business process model and notation*) i SPEM 2.0 (*software and systems process engineering metamodel specification*), totes dues actualment responsabilitat de l'OMG. Els diagrames d'activitat d'UML també es poden usar per a descriure aquest tipus de processos, encara que de manera una mica més genèrica.

Webs recomants

Pots trobar tota la informació de BPMN 2.0 i de SPEM 2.0 en els seus webs oficials. Per BPMN 2.0 a http://www.bpmn.org i a http://www.omg.org/spec/BPMN/2.0/ i per a SPEM 2.0 a http://www.omg.org/spec/SPEM/2.0/.

Els models de negoci d'alt nivell es poden usar per a múltiples funcions que cada vegada estan cobrant més rellevància en MDE, com són els de l'enginyeria de processos i l'adquisició basada en models.

- Dins del que es denomina *enginyeria de processos de negoci*, disposar de models de processos permet fer certes anàlisis molt útils per a estudiar les seves prestacions (*performance*), detectar colls d'ampolla o fins i tot situacions de bloquejos (*deadlocks*).

- Els models dels processos de negoci permeten descriure els requisits del sistema que es pretén construir amb un nivell d'abstracció adequat per a servir com a "plecs de requisits" en els processos d'adquisició de sistemes d'informació. En aquest context, una empresa o administració genera els models de negoci del sistema que pretén adquirir, i els proveïdors usen aquests models com a base per a desenvolupar les aplicacions en les seves plataformes. Aquesta pràctica ha donat lloc al que es coneix com a *model-based acquisition* (MBA).

A més dels acrònims esmentats fins al moment, és possible trobar-ne molts més en la bibliografia relacionada amb l'enginyeria del programari dirigida per models. En aquesta secció se'n destaquen dos: ADM i MDI.

Architecture-driven modernization (**ADM**) és una proposta de l'OMG per a implementar pràctiques d'enginyeria inversa, utilitzant models.

Web recomanat

Informació detallada sobre la proposta ADM de l'OMG, incloent-hi els estàndards relacionats, eines i casos d'ús d'èxit en empreses; es pot consultar a http://adm.omg.org/

L'objectiu d'ADM és extreure models a diferents nivells d'abstracció d'un codi o aplicació existent, amb l'ànim d'extreure la informació independent dels llenguatges i plataformes tecnològiques utilitzades. Tenir aquesta informació permetrà utilitzar després tècniques d'MDE per a analitzar el sistema, generar-lo en plataformes diferents, etc. La paraula *modernització* es deu a l'objectiu que perseguia al principi ADM, que no era sinó migrar aplicacions existents envers noves plataformes.

Exemples

El Departament de Defensa nord-americà es va veure en la necessitat de migrar els programes dels seus avions de combat, que usaven microprocessadors dels anys setanta, i per als quals no havia recanvis. Igualment, la NASA el 2002 usava processadors 8086 en molts dels seus equips de l'Space Shuttle Discovery, i temia que un procés de migració manual del programari pogués ocasionar una pèrdua de qualitat significant i uns costos desorbitats, a part del temps i esforç necessari per a repetir totes les proves requerides per a un programari crític com el de control d'avions de combat o aeronaus tripulades. Fins i tot si la migració fos un èxit, dins d'uns quants anys es tornarien a veure en la mateixa situació –tret que ideessin una nova manera de migrar els seus sistemes. La solució va venir de la idea d'extreure automàticament models independents de la plataforma dels seus programes, i definir transformacions de models que "compilessin" aquests models (independents de la plataforma i de qualsevol tecnologia) en el codi apropiat als nous processadors i sistemes per utilitzar.

Model driven-interoperability (**MDI**) és una iniciativa per a implementar mecanismes d'interoperabilitat entre serveis, aplicacions i sistemes usant models i tècniques d'MBE.

Per ***interoperabilitat*** s'entén l'habilitat de dues o més entitats, sistemes, eines, components o artefactes per a intercanviar informació i fer un ús adequat de la informació intercanviada per a treballar de manera conjunta.

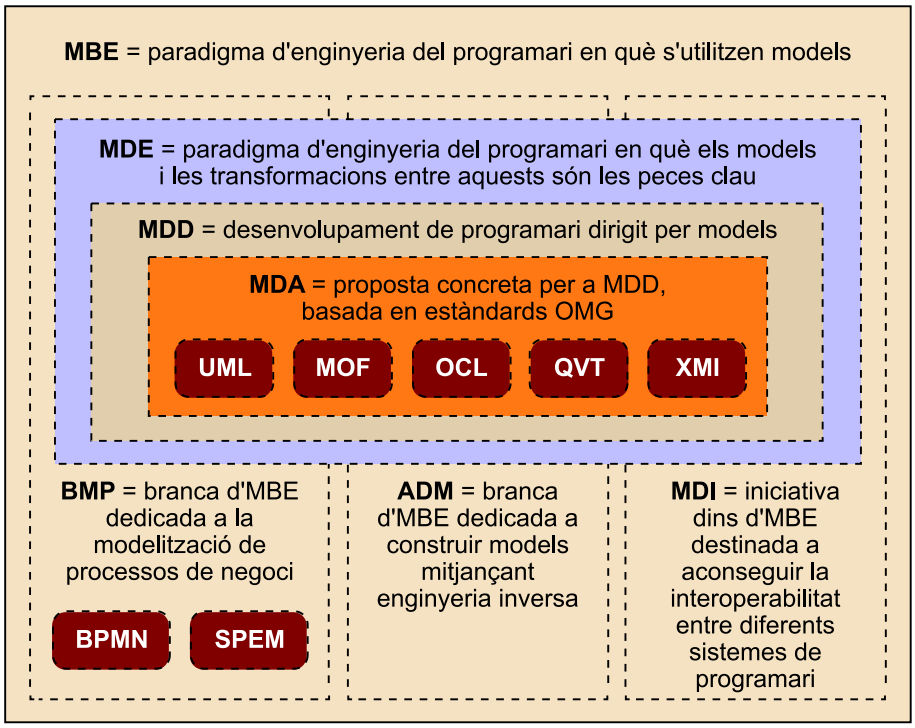
Encara que la interoperabilitat sempre ha estat un requisit essencial en qualsevol sistema format per dos o més components, aquest problema mai no havia estat resolt satisfactòriament per les dificultats per a resoldre les diferències entre els sistemes, la falta d'acord en els estàndards per utilitzar de manera conjunta i la falta de mecanismes i eines apropiades.

Millores d'interoperabilitat

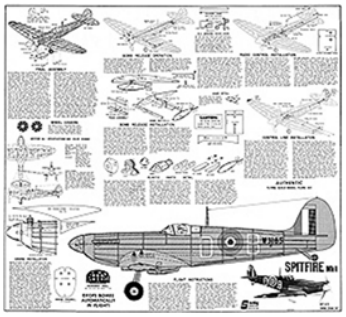
La comunitat MDE ha començat a treballar en aquests temes, organitzant tallers i conferències internacionals en què experts en diferents matèries hi poden discutir i tractar d'arribar a acords. És interessant el plantejament del problema presentat per Jean Bezivin en la seva presentació sobre MDI. Per la seva banda, l'OMG ha creat un grup de treball (Model-Interchange Working Group) per a resoldre els problemes d'interoperabilitat entre les diferents eines que treballen amb models, i en particular les eines UML. Els principals fabricants d'eines de modelització formen part d'aquest grup, amb l'objectiu de conjuminar esforços i ser compatibles entre si.

A manera il·lustrativa, la figura 3 mostra tots els conceptes definits en aquesta secció.

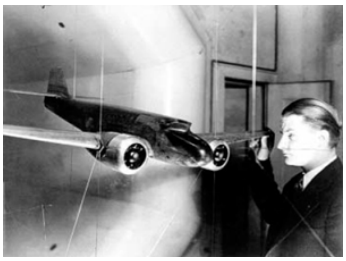
Figura 3. Terminologia relacionada amb el desenvolupament de programari dirigit per models



1.4. Els models com a peces clau d'enginyeria



Pensem un moment en com es dissenyen i desenvolupen els sistemes complexos en altres enginyeries tradicionals, com l'enginyeria civil, l'aeronàutica, l'arquitectura, etc. Aquest tipus de disciplines fa segles que construeixen amb èxit gratacels, coets espacials o increïbles ponts penjants, la complexitat dels quals és sens dubte tan elevada o més que la dels nostres sistemes de programari.



Una cosa que diferencia aquestes disciplines de l'enginyeria del programari, almenys en el procés de disseny i construcció, és que tenen llenguatges i notacions per a produir models d'alt nivell, independents de les plataformes i tecnologies de desenvolupament i de la implementació final, i amb processos ben definits que són capaços de transformar aquests models en les construccions finals d'una manera **predictible**, **fiable** i **quantificable**.

> Si l'arquitecte que dissenya un gratacel de més de 100 pisos hagués de tenir en compte en el disseny de l'edifici detalls de molt baix nivell com la fusta dels marcs de les portes, el tipus d'aixeteria que caldrà usar o el tipus de clau amb el qual es clavaran els quadres, i a més haver de dissenyar i construir aquests marcs, aixetes i claus, el projecte seria impracticable.

També és important considerar el fet que els enginyers i arquitectes tradicionals no usen només un plànol del que han de construir, sinó que normalment en desenvolupen uns quants, cadascun expressat en un llenguatge diferent i a un nivell d'abstracció diferent. El tipus de plànol dependrà del tipus de propietats que vulguin especificar, i també del tipus de preguntes que volen ser capaços de respondre amb aquest plànol.

Els tipus de plànols poden usar diferents llenguatges, notacions i fins i tot materials. Per exemple, per a un avió es pot usar un plànol per a l'estructura bàsica, un altre amb l'especejament de les parts mecàniques a escala, un altre per a les connexions elèctriques, una maqueta en plàstic per a estudiar-ne l'aerodinàmica en el túnel de vent, un prototip en metall per a estudiar-ne l'amortiment i càrregues, un model de mida reduïda per a mostrar als clients, etc.

Cada model (plànol, maqueta, prototip) és el més adequat per a una funció, que és el que es denomina el **propòsit del model**, i cadascun serà útil en un moment diferent del procés de desenvolupament.

Així, el plànol amb l'estructura servirà per a construir la maqueta i posteriorment per a guiar la resta dels plànols. El model a escala servirà durant la fase d'anàlisi per a comprovar si satisfà els requisits dels clients i del departament de màrqueting. La maqueta permetrà estudiar els aspectes aerodinàmics i refinar el model abans de la fase de disseny detallat. Els plànols de l'estructura i especejament serviran als mecànics per a construir l'avió, etc.

Una cosa molt important que s'ha d'assenyalar és que els diferents plànols no són completament independents, sinó que tots representen el mateix sistema, encara que des de diferents punts de vista i a diferents nivells d'abstracció.

No obstant això, aquesta manera de procedir d'altres enginyeries no és comuna en el desenvolupament de programari. Els enginyers de programari no solen produir models d'alt nivell dels processos de negoci dels seus sistemes d'informació, ni tan sols de l'arquitectura detallada de les seves aplicacions. Això fa que l'única documentació del sistema d'informació que ha adquirit que queda a l'empresa és el codi final. Fins i tot en aquells casos en els quals l'arquitecte de programari va produir els models de l'estructura de l'aplicació (usant, per exemple, UML), durant la resta dels processos de desenvolupament (codificació, proves i manteniment) normalment s'alteren molts detalls que no es reflecteixen en aquests "plànols". Això fa que els models originals quedin ràpidament desactualitzats i, per tant, inútils.

Aquests problemes són els que van fer sorgir a començaments dels anys 2000 un moviment que promulgava l'ús dels models com a peces clau del desenvolupament de programari, i el primer exponent del qual va ser la proposta denominada MDA (*model-driven architecture*), de l'organització OMG (Object Management Group). Aquesta proposta, que detallarem en les seccions següents, promulgava l'ús de diferents tipus de models, cadascun a diferents nivells d'abstracció.

Nota

OMG és un consorci industrial, format per més de 700 empreses, el propòsit de les quals és definir estàndards per a la interoperabilitat de sistemes de programari. Inicialment centrada en sistemes orientats a objectes (p. ex., CORBA), avui se centra en estàndards per a modelització de sistemes i processos (UML, MOF, OCL, BPMN, QVT, etc.).

La idea bàsica d'MDA consisteix a elaborar primer models de molt alt nivell, denominats *models independents de les plataformes* o PIM (*platform-independent models*) i completament independents de les aplicacions informàtiques que els implementaran o les tecnologies usades per a desenvolupar-los o implementar-los. MDA defineix llavors alguns processos per a anar refinant aquests models, particularitzant-los progressivament en models específics de la plataforma que s'ha d'usar o PSM (*platform-specific models*) cada vegada més concrets tal com es van determinant els detalls finals.

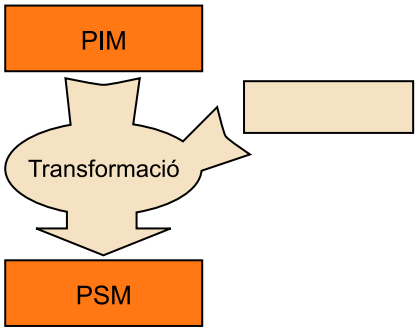
El més interessant d'MDA és que les plataformes que detallen els serveis per usar (programari intermediari, sistemes operatius, maquinari o algunes de més abstractes com les que proporcionen requisits de seguretat) estan també descrites mitjançant models, i que els processos que van refinant els models d'alt nivell fins a arribar a la implementació final estan descrits per transformacions de models, que no són sinó models al seu torn.

Aquesta idea es coneix com el *patró MDA*, que gràficament es mostra en la figura 4, on es pot veure un model PIM que es transforma en un model PSM mitjançant una transformació automatitzada. Aquesta transformació té com a entrada un altre model, que pot ser el de la plataforma concreta en què s'ha de generar el model PSM, el de certs paràmetres de configuració de l'usuari, etc. Aquest model d'entrada "parametritza" la transformació, i està representat en la figura 4 per una caixa buida.

Vegeu també

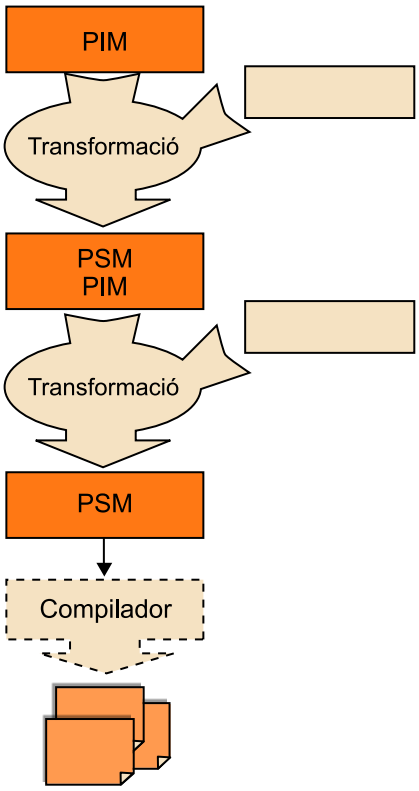
Els elements d'aquest patró es veuran amb més detall en l'apartat "Transformacions de models".

Figura 4. El patró MDA



Utilitzant aquesta idea, en MDA el procés de desenvolupament del programari és, per tant, un procés iteratiu de transformació de models (figura 5).

Figura 5. El procés de desenvolupament en MDA



En la figura 5 podem observar com cada pas transforma un PIM del sistema considerat a un cert nivell en un PSM del nivell següent, fins que s'aconsegueix la implementació final del sistema, amb la particularitat que cada PSM resultat d'una transformació es pot convertir en el PIM per a la transformació següent (pel que fa a una altra plataforma o nivell d'abstracció).

És important assenyalar que, en aquest context, una implementació no és més que el model final d'un sistema que conté tota la informació necessària per a la implantació i posada en funcionament, és a dir, una altra forma de PSM; potser el PSM de més baix nivell.

Exemples

A partir d'un model amb l'especificació bàsica del disseny d'una base de dades i d'una interfície gràfica d'accés és possible generar tot el codi de l'aplicació en plataformes diferents, usant diferents tecnologies de bases de dades i plataformes de programari. Un altre exemple de l'ús d'aquesta cadena de transformacions MDA es dóna en el que es coneix com a *model-driven web engineering*, en què a partir del PIM d'un sistema web compost per un model de les dades persistents de l'aplicació (el model del *contingut*), d'un model de navegació, i d'un model de presentació abstracte, és possible definir transformacions automàtiques a plataformes concretes de components (com EJB o .NET) que usin interfícies d'usuari basades en diferents tecnologies (com PHP o Silverlight). Això és possible amb algunes de les propostes, com UWE o WebML.

Aquest plantejament d'usar models independents de la tecnologia presenta nombrosos avantatges. Entre aquests, que les empreses poden tenir models d'alt nivell dels seus processos de negoci, de la informació que manegen i dels serveis que ofereixen als seus clients i els que requereixen dels seus proveïdors, de manera independent de la tecnologia que els sustenti i de les plataformes de maquinari i programari que usin en un moment donat. D'aquesta manera,

aquests models es converteixen en els veritables actius de les companyies, amb una vida molt superior als sistemes d'informació que els implementen, ja que les tecnologies evolucionen molt més ràpid que els processos de negoci de les empreses.

El model de negoci d'una companyia que ofereix servei de traducció jurada de documents es pot descriure independentment de si el sistema d'informació que el suporta va ser desenvolupat per a un ordinador central en els anys seixanta, usant una arquitectura client-servidor en els vuitanta, mitjançant una aplicació web el 2000 o en el "núvol" actualment.

D'altra banda, també és possible reutilitzar els models de les plataformes tecnològiques o d'implementació, i també de les transformacions que converteixen certs models d'alt nivell a aquestes plataformes.

Un altre avantatge molt important és que aquest enfocament MDA permet el tractament unificat de tots els artefactes de programari que intervenen en el disseny, desenvolupament, proves, manteniment i evolució de qualsevol sistema, ja que tots passen a ser models, fins i tot les transformacions mateixes o el codi de l'aplicació.

Lectura recomanada

J. Bezivin (2005). "The Unification Power of Models". *SoSym* (vol. 4, núm. 2, pàg. 171-188).

En MDD el procés de desenvolupament de programari es converteix, per tant, en un procés de modelització i transformació, mitjançant el qual el codi final es genera majorment de manera automàtica a partir dels models d'alt nivell, dels models de les plataformes destinació i de les transformacions entre aquests.

Nota

Del lema dels anys dels noranta "Everything is an object" es passa el 2003 a "Everything is a model."

A més d'aconseguir elevar el nivell d'abstracció notablement enfront dels llenguatges de programació convencionals i els seus processos de desenvolupament habituals, també s'obtenen millores substancials en altres fronts:

1) En primer lloc, la gestió dels canvis en l'aplicació, i també el seu manteniment i evolució es modularitzen d'una manera estructurada. Un canvi en la tecnologia d'implementació de l'aplicació només afectarà normalment les plataformes de nivell més baix i les transformacions que generen els models PSM a aquest nivell, i deixarà inalterats els models superiors (models de negoci, requisits de seguretat, etc.). Aquests models d'alt nivell tindran una vida més llarga i es tornen menys dependents en els canvis en els nivells inferiors.

2) D'altra banda, el fet que la propagació dels canvis i la generació de codi les duguin a terme les transformacions de model fan que la regeneració de l'aplicació es pugui fer de manera gairebé automàtica, estalviant costos i esforç, a més de poder garantir-se més qualitat.

Fent una analogia amb la programació convencional, podem considerar els models d'alt nivell com els programes, i la implementació de l'aplicació com el codi màquina. Les transformacions de models són les que serveixen per a compilar els programes i generar el codi màquina corresponent. No obstant això, el plantejament del procés de desenvolupament de programari en termes de models a diferents nivells d'abstracció i transformacions entre aquests ens permet fer un salt qualitatiu, i eliminar molts dels problemes que plantejaven els llenguatges de programació quant a expressivitat, insuficient nivell d'abstracció, complexitat accidental, etc. Això permetrà a l'enginyer de programari centrar-se en l'especificació dels models en el domini del problema en comptes de fer-ho en el domini de la solució, i a un nivell d'abstracció adequat al problema que pretén resoldre. Per *domini del problema* entenem l'espai o àmbit d'aplicació del sistema. Per *domini de la solució* s'entén el conjunt de llenguatges, metodologies, tècniques i eines que s'usen per a resoldre un problema concret.

3) Finalment, tenir diferents models del sistema, cadascun centrat en un punt de vista determinat, i a un nivell d'abstracció ben definit, obre un ventall de possibilitats i línies d'actuació molt interessants, que discutim en les seccions següents.

1.5. Tipus de models en MDA

Com s'ha explicat abans en la secció 1.4, la proposta d'MDA s'organitza bàsicament entorn de models independents de la plataforma (PIM, *platform independent models*), models específics de la plataforma (PSM, *platform specific models*) i transformacions entre models.

Vegeu també

Vegeu la figura 4.

Un **model independent de la plataforma**, o PIM, és un model del sistema que concreta els seus requisits funcionals en termes de conceptes del domini, i és independent de qualsevol plataforma.

Generalment, la representació d'un model PIM està basada en un llenguatge específic per al domini modelitzat, en què els conceptes representats exhibeixen un cert grau d'independència respecte a diverses plataformes, ja siguin plataformes tecnològiques concretes (com CORBA, .NET o J2EE) o plataformes més abstractes (com per exemple requisits de seguretat o de fiabilitat). És així com MDA pot assegurar que el model independent de la plataforma (PIM) sobreviurà als canvis que es produeixin en futures plataformes de tecnologies i arquitectures de programari.

Exemple

El model de la cadena de muntatge mostrat en la figura 8 és un exemple de PIM.

Un **model específic per a una plataforma** (PSM) és un model resultat de refinar un model PIM per a adaptar-lo als serveis i mecanismes oferts per una plataforma concreta.

A part de models PIM i PSM, en MDA també són importants les plataformes, que al seu torn es descriuen com a models.

En MDA, una **plataforma** és un conjunt de subsistemes i tecnologies que descriuen la funcionalitat d'una aplicació per mitjà d'interfícies i patrons específics, i facilita que qualsevol sistema que hagi de ser implementat sobre aquesta plataforma pugui fer ús d'aquests recursos sense tenir en consideració aquells detalls que són relatius a la funcionalitat oferta per la plataforma concreta.

Partint de la representació d'un PIM i donat un model de definició d'una plataforma (PDM, *platform definition model*), el PIM es pot traduir a un o més models específics de la plataforma (PSM) per a la implementació corresponent usant novament llenguatges específics del domini, o llenguatges de propòsit general com Java, C#, Python, etc.

Al principi, els perfils UML (*UML profiles*) van ser l'alternativa més utilitzada com a llenguatge de modelització per a crear models PIM i PSM, però ara han guanyat en acceptació nous llenguatges definits a partir de llenguatges de metamodelització, com MOF o Ecore.

Vegeu també

Els perfils UML s'estudien amb detall en l'apartat 3, en parlar de com es poden definir llenguatges específics de domini.

Un **perfil UML** es defineix com una extensió d'un subconjunt d'UML orientada a un domini. Es descriu a partir d'una especialització d'aquest subconjunt i utilitzant els conceptes que incorpora el mecanisme d'extensió d'UML: **estereotips**, **restriccions** i **valors etiquetats**. Com a resultat s'obté una variant d'UML per a un propòsit específic.

Actualment hi ha perfils adoptats per OMG per a plataformes com CORBA, Java, EJB o C++. Més endavant explicarem amb detall els perfils d'UML, que permetran definir tant la sintaxi com la semàntica d'un llenguatge, mitjançant un refinament de les d'UML. A més, l'ús d'icones específiques farà possible dotar els models de notacions gràfiques més usables i atractives.

Exemple

Un exemple d'icones específiques és mostrat en la figura 9.

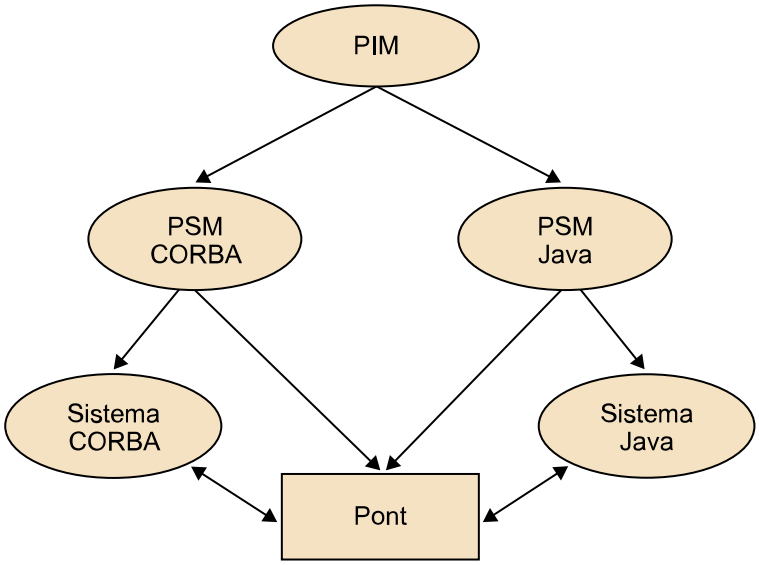
1.6. El procés de desenvolupament MDA

Tal com esmentem en la secció "Conceptes bàsics", en MDA el procés de desenvolupament de programari és un procés de transformació de models iteratiu en el qual en cada pas es transforma un PIM del sistema considerat a un cert nivell en un PSM del nivell següent, fins que s'aconsegueix la implementació final del sistema (vegeu la figura 5). Les transformacions de models són les que s'encarreguen de "dirigir" el procés. Encara que aquest és el cas més bàsic, hi ha altres processos MDA d'interès, com els que hem esmentat anteriorment de modernització de sistemes (ADM) i d'interoperabilitat basada en models (MDI).

Vegeu també

Els procés ADM i MDI van ser introduïts en la secció "Terminologia".

Figura 6. Ponts d'interoperabilitat en MDA



Un altre cas interessant en què els models poden proporcionar importants beneficis és en la integració de sistemes. Generalment la implementació d'un sistema requereix l'ús i cooperació entre diverses tecnologies de programari per a assolir la funcionalitat especificada. Aquest és el cas de les aplicacions que s'implementen reutilitzant components o sistemes externs. Així, la integració entre diferents tecnologies pot ser especificada a escala de models en el context d'MDA mitjançant la definició explícita de ponts (*bridges*), com s'observa en la figura 6.

Un **pont** (*bridge*) es defineix com un element de modelització que permet comunicar i integrar sistemes heterogenis entre si.

En MDA, la implementació de ponts se simplifica enormement, i les eines de transformació s'encarreguen de generar els ponts de manera automàtica unint les diferents implementacions i els diferents models de les plataformes.

1.7. Reptes actuals d'MDD i MDA

Els primers resultats de l'adopció de les pràctiques d'MDD són prometedors, i estan demostrant ser realment efectius en aquells casos en els quals s'ha aplicat adequadament. Per descomptat, com qualsevol altra tecnologia o aproximació, MDA no està exempta de riscos. A part d'introduir nombrosos avantatges i oportunitats, és important ser conscient dels reptes que haurem d'afrontar en el nostre negoci en plantejar un nou desenvolupament adoptant aquesta proposta.

- Costos en l'adopció de les tècniques i eines MDD, per la corba d'aprenentatge i inversió que representen per a qualsevol organització.

- Problemes d'adopció per part dels equips de desenvolupament actuals, en tractar-se de pràctiques bastant diferents de les tradicionals.

- Problemes a escala organitzativa, ja que molts gestors no estan disposats a donar suport a inversions el benefici de les quals a molt curt termini no és trivial (encara que a llarg termini puguin millorar significativament la productivitat de l'empresa).

- Falta de maduresa en algunes de les eines MDD (editors de models, generadors de codi, màquines de transformació), principalment a causa de la implantació recent d'aquesta disciplina.

- Falta de bones pràctiques i de processos de desenvolupament MDD genèrics, la qual cosa no permet adoptar aquesta pràctica de manera unificada i *off-the-shelf*.

Web recomanat

L'adopció d'MDA té ja nombrosos casos d'èxit en empreses i organitzacions, incloent-hi Daimler Chrysler, Lockheed Martin, Deutsche Bank, ABB, National Cancer Institute, o Credit Suisse. La descripció de molts d'aquests casos es pot consultar a http://www.omg.org/mda/products_success.htm.

A manera de resum, el quadre següent mostra algunes circumstàncies en les quals és convenient aplicar les tècniques MDD en una empresa, enfront d'aquelles que potser ho desaconsellen.

És convenient aplicar MDD quan...	No és convenient aplicar MDD quan...
El producte es desplegarà en múltiples plataformes o interessa deslligar-lo d'una tecnologia concreta perquè tindrà una vida útil perllongada.	El producte es desplegarà en una sola plataforma que està prefixada per endavant i la seva vida útil serà molt curta.
L'equip pot invertir esforços a adaptar o desenvolupar eines pròpies per a obtenir beneficis a mitjà/llarg termini.	El projecte necessita disposar d'una tecnologia madura per a obtenir resultats immediats.
Es disposa d'un equip amb experiència en MDD o bé recursos i temps per a formar-se.	No es disposa d'experiència en MDD ni oportunitats per a formar-se.
Els gerents i directius de l'empresa donen suport al canvi de paradigma de desenvolupament.	Hi ha molt recel o fins i tot rebuig per part de la direcció de l'organització.

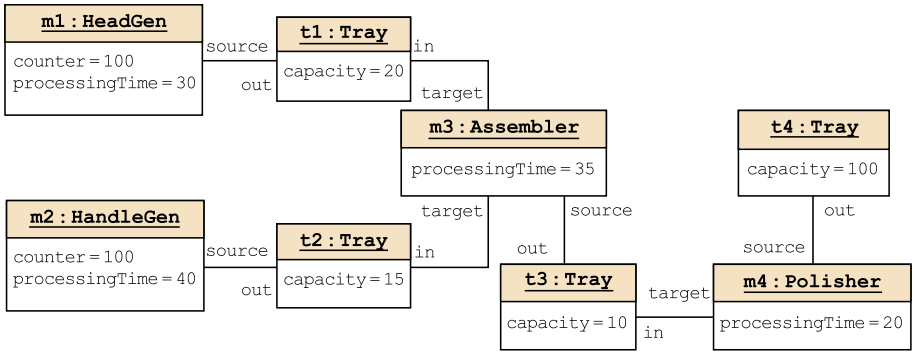
Una cosa que en general està donant molt bon resultat en moltes empreses és l'adopció de processos MDD en alguns equips de desenvolupament i per a projectes molt concrets, a manera d'experiment controlat en què se'n poden provar els resultats i valorar aquestes pràctiques.

1.8. Un cas d'estudi

Per il·lustrar els conceptes vistos en aquest capítol, i com a fil argumental per a la resta del mòdul, modelitzarem un sistema, concretament una cadena de muntatge d'una fàbrica de martells (d'ara endavant ens hi referirem com a "CadenaDeMuntatge"). El sistema té quatre tipus de màquines: generadors de mànecs (`HandleGen`), generadors de capçals (`HeadGen`), acobladores (`Assembler`) i polidores (`Polisher`). Les dues primeres generen les peces bàsiques, que han de ser acoblades posteriorment per a formar martells, i que finalment

seran embellits en la polidora. Les màquines es connecten mitjançant safates (`Tray`), des de les quals prenen les peces d'entrada, i hi col·loquen les peces produïdes. Cada màquina triga un temps a processar una peça, i cada safata té una capacitat limitada. Una disposició possible de la cadena de muntatge es mostra en el diagrama de classes de la figura 7.

Figura 7. Un model d'una cadena de muntatge de martells

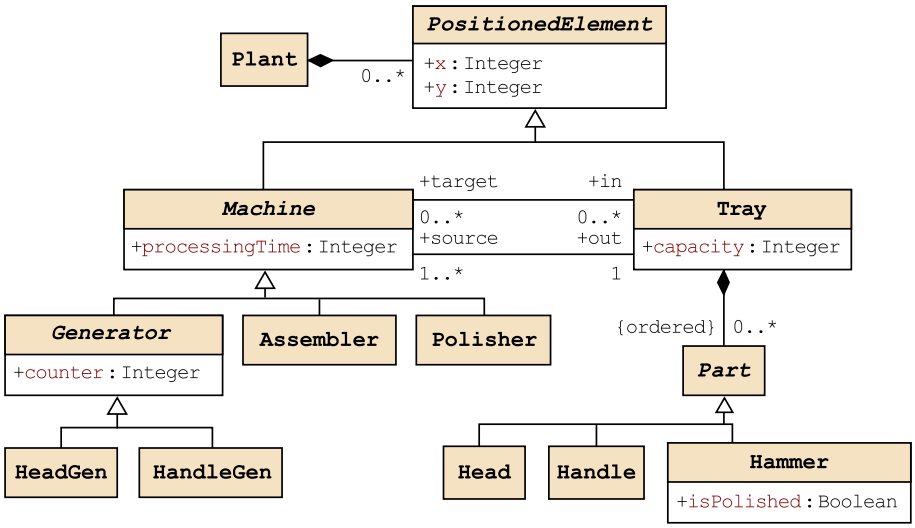


La de la figura 7 és una possible configuració del sistema, encara que se'n podrien establir d'altres depenent del nombre de màquines generadors i acobladores que s'utilitzin per a accelerar el procés, la qualitat de les màquines (hi ha màquines en el mercat amb menys temps de processament, però costen més), la capacitat de les safates (de nou, com més capacitat més preu), etc.

El que es busca amb aquest sistema és maximitzar la producció tractant d'optimitzar els costos i els temps.

Com podem observar, el model mostrat en la figura 7 és un model PIM, independent de cap plataforma concreta. Ens podem preguntar quin és el seu metamodel, que no és sinó un altre model, però que conté els conceptes del domini del problema i les relacions entre aquests. Un possible metamodel d'aquest model és el que es mostra en la figura 8.

Figura 8. Metamodel de la cadena de muntatge de martells



En aquest cas el model i el seu metamodel estan expressats en UML.

El metamodel descriu les classes dels objectes que formen el model i les relacions entre aquests, i el model descriu els elements que formen part del sistema que volem representar.

UML i les restriccions OCL

En general, UML no serà suficient per a definir correctament metamodels, sinó que caldrà incloure algunes restriccions d'integritat. Per exemple, el nombre de parts que conté una safata ha de ser sempre més petit que la seva capacitat. En UML, aquestes restriccions d'integritat s'expressen fent ús d'OCL, un llenguatge que estudiarem en l'apartat 2 d'aquest mòdul.

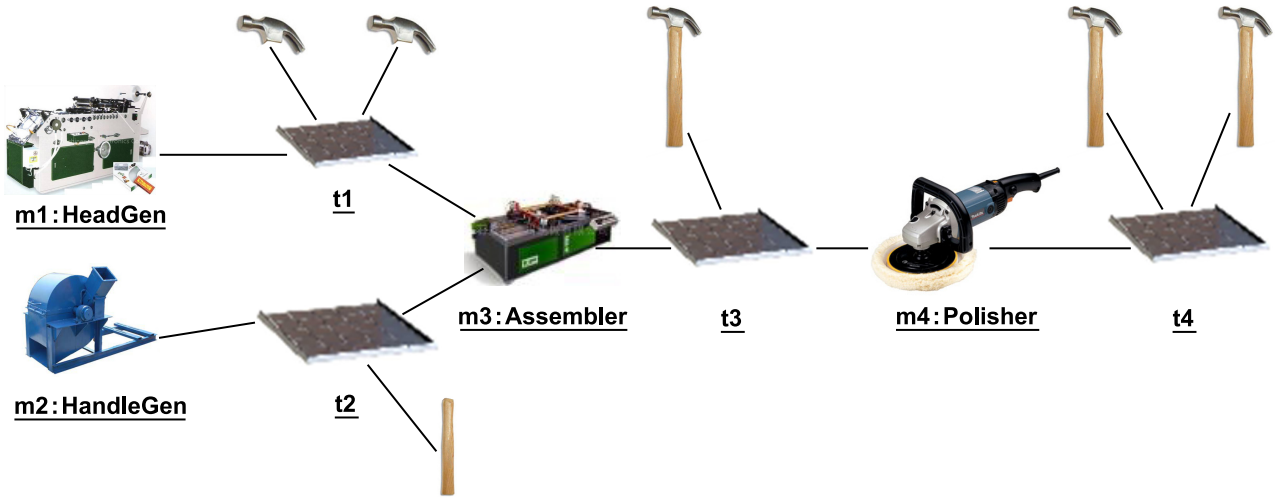
El propòsit del model és oferir una descripció d'alt nivell dels elements bàsics del sistema i la seva estructura, amb l'objectiu que ho comprenguin els usuaris finals. No obstant això, aquest model ignora molts altres aspectes del sistema, perquè no són rellevants des d'aquest punt de vista.

El model no descriu de manera explícita el comportament del sistema, ni la tecnologia amb la qual està implementat, ni l'estructura de la base de dades que emmagatzema la informació que volem que sigui persistent, etc.

Altres models seran els encarregats d'especificar aquests i altres aspectes, com veurem més endavant.

Usant un perfil UML adequat és possible expressar el model mostrat en la figura 7 com apareix en la figura 9 (hem inclòs algunes peces en les safates per mostrar com es veurien amb aquesta notació, representant, no el model inicial del sistema, sinó un model en un instant donat en el temps).

Figura 9. El model de la cadena de muntatge expressat amb icones



1.9. Conclusions

En aquest capítol hem presentat una descripció de l'estat actual del desenvolupament de programari dirigit per models, especialment en el context d'MDA. Per a això, hem analitzat els conceptes i mecanismes que aporta aquesta proposta i com pot beneficiar el procés de desenvolupament de grans sistemes de programari per les empreses del sector.

Una vegada hem vist en aquest capítol els conceptes generals d'MDA, en els següents estudiarem detalladament alguns dels llenguatges i tecnologies associades: OCL per a especificar amb precisió els models UML, les tècniques de metamodelització per a definir llenguatges específics de domini, i els llenguatges i eines de transformació de models.

2. El llenguatge de restriccions i consultes OCL

Vegeu també

Podeu trobar una introducció a OCL en l'assignatura *Enginyeria de requisits* del grau d'Enginyeria Informàtica.

En l'apartat anterior hem comentat la importància que té per a MDE el que els models siguin **precisos** i **complets**. La precisió implica que la interpretació no admeti cap ambigüitat, la qual cosa garanteix que tots els que utilitzaran els models (dissenyadors, desenvolupadors, analistes, usuaris finals i, sobretot, altres programes!) els interpretaran de la mateixa manera. Ser complets implica que han de contenir tots els detalls rellevants per a representar el sistema des del punt de vista adequat.

Encara que hi ha notacions, tant gràfiques com textuals, per a això, quan es pensa en representació de models, sobretot de sistemes de programari, se sol pensar en notacions gràfiques com UML. Aquestes notacions són molt apropiades per a representar models d'una manera visual i atractiva, però no posseeixen l'expressivitat suficient per a descriure tota la informació que ha de contenir un model.

El model de la figura 8 de l'apartat anterior, que descriu el metamodel de la cadena de muntatge, no descriu algunes propietats molt importants del sistema, com que el nombre de parts que conté una safata ha de ser sempre més petit que la seva capacitat, o que els comptadors de peces han de ser positius. Aquest tipus de condicions no es poden expressar només amb la notació que proporciona UML.

Tradicionalment, en el cas d'UML, el que se sol fer és completar els models amb descripcions en llenguatge natural sobre aquest tipus de condicions i qualsevol altra informació addicional sobre el model que sigui rellevant.

Exemples

La semàntica del model i el seu entorn, les restriccions d'integritat, l'especificació dels valors inicials dels atributs, el comportament de les operacions, etc.

Consulta recomanada

Axel van Lamsweerde proposa una definició de llenguatge formal a: "Formal specification: a roadmap". Proceedings of the Conference on The Future of Software Engineering (FOSE'00) (pàg. 147-159). ACM, NY, 2000. http://doi.acm.org/10.1145/336512.336546.

El problema és que les descripcions en llenguatge natural, encara que són fàcils d'escriure i llegir per qualsevol persona, són normalment ambigües, i per descomptat no són manipulables per altres programes, que és el que precisament requereix MDE. D'altra banda, també sabem que existeixen els llenguatges denominats *formals*, que no tenen ambigüitats però que són difícils d'usar per persones que no tinguin un cert bagatge matemàtic, tant a l'hora d'escriure especificacions de sistemes com de comprendre les escrites per altres. Una altra opció, per descomptat, és usar el codi font, que no és sinó un model final del programari que conté tots els detalls. No obstant això, el baix nivell de detall i l'elevada complexitat no ho converteixen en la millor opció per a comprendre el sistema i raonar-hi.

OCL (*object constraint language*) és un llenguatge textual, amb base formal, i que a més posseeix mecanismes i conceptes molt propers als d'UML, la qual cosa en facilita l'ús pels modelitzadors. Per exemple, l'expressió OCL següent indica que el nombre de parts que hi ha en una safata en qualsevol moment ha de ser inferior a la seva capacitat:

Figura 9. El model de la cadena de muntatge expressat amb icones

```
context Tray inv NoOverflow: self.part->size() <= self.capacity
```

OCL és un llenguatge formal, basat en teoria de conjunts i lògica de primer ordre, que va ser desenvolupat inicialment com a part del mètode de disseny de sistemes denominat *Syntropy*, de Steve Cook i John Daniels, i posteriorment va ser adoptat per la Divisió d'Assegurances d'IBM com a llenguatge precís de modelització dels seus processos de negoci.

Consulta recomanada

Syntropy és un mètode de disseny de sistemes orientat a objectes, sobre els quals es va basar UML. Està descrit en el llibre S. Cook, J. Daniels (1994) *Designing Object Systems: Object-Oriented Modelling with Syntropy*, Prentice Hall. http://www.syntropy.co.uk/syntropy/.

Quan UML va necessitar un llenguatge per a expressar restriccions d'integritat sobre els seus models, la comunitat MDA va veure OCL com el candidat idoni. Els seus mecanismes per a navegar models encaixaven de manera natural amb l'estructura d'UML, i tots dos llenguatges es complementaven molt bé. Una de les principals aportacions de l'OMG va ser la integració completa d'OCL amb UML i amb MOF a l'hora d'incorporar-lo al seu catàleg d'estàndards de modelització i d'MDE, fent compatibles els seus sistemes de tipus i les maneres d'assignar noms als seus elements. A més, cadascun d'aquests llenguatges aporta els seus avantatges en ser combinats, i s'arriba al punt que en el context d'MDA no poden viure l'un sense l'altre: per a obtenir un model complet, són necessaris tant els diagrames UML com les expressions OCL.

Web recomanat

OCL 2.0 va ser adoptat el 2003 pel grup OMG com a part d'UML 2.0. L'estàndard d'OCL publicat per l'OMG està disponible a www.omg.org/spec/ocl/. La versió descrita aquí correspon a OCL 2.3.1.

2.1. Característiques d'OCL

En primer lloc, OCL és un llenguatge **fortament tipat**. En altres paraules, tota expressió OCL és d'un tipus determinat. Per a això, els sistemes de tipus d'UML i OCL són completament compatibles: el conjunt de tipus bàsics d'OCL es correspon amb els que defineix UML (`Integer`, `String`, `Set`, `Sequence`, etc.), cada classe UML defineix un tipus OCL, i l'herència en UML s'interpreta com a subtipatge en OCL.

Una altra característica interessant d'OCL és que és un llenguatge d'especificació que **no té efectes laterals**. Això vol dir que l'avaluació de qualsevol expressió OCL no pot modificar l'estat del model, sinó només consultar-lo i retornar un valor (d'un cert tipus, precisament el tipus de l'expressió OCL). Per tant, no es pot considerar OCL com un llenguatge de programació: les seves sentències no són executables i només permeten consultar l'estat del sistema, avaluar una expressió o afirmar una cosa sobre aquest[2].

[(2)]Per exemple, un invariant que ha de ser sempre cert, o l'estat del sistema després d'una operació

És important assenyalar també que l'avaluació d'una expressió OCL se suposa **instantània**. Això vol dir que els estats dels objectes d'un sistema no poden canviar mentre s'avalua una expressió.

Finalment, OCL té diverses eines que en permeten comprovar les especificacions sobre diferents tipus de models. Entre aquestes destaquem les següents:

- ***Dresden OCL toolkit*** proporciona un conjunt d'eines per a l'anàlisi sintàctica i avaluació de restriccions OCL en models UML, EMF i Java, alhora que proporciona eines per a la generació de codi Java/AspectJ i SQL. És possible utilitzar les eines del Dresden OCL com una biblioteca en altres projectes o com un connector Eclipse per a manipular expressions OCL.

- **USE**, eina desenvolupada per la Universitat de Bremen, està dissenyada per a especificar tant models UML com les seves restriccions d'integritat usant OCL, i

- **MDT/OCL** per a models Ecore d'Eclipse.

Nota

El portal OCL ofereix referències a les diferents eines per a OCL disponibles actualment, i també referències a treballs i altres pàgines web en què les eines són comparades i analitzades (http://st.inf.tu-dresden.de/ocl/).

Web recomanat

Més informació sobre USE: http://www.sourceforge.net/apps/mediawiki/useocl/.

Web recomanat

Pàgina web d'MDT/OCL: www.eclipse.org/projects/project.php?id=modeling.mdt.ocl.

No obstant això, no gaires eines de modelització (editors de models, entorns de desenvolupament de codi, etc.) donen suport actualment a l'especificació i validació d'expressions OCL. I entre aquelles que ho permeten, molt poques són capaces d'utilitzar-lo en tota la seva potència (per exemple, per a la generació de codi). De totes maneres, la necessitat d'utilitzar els models de manera precisa està fent madurar les eines OCL ràpidament.

2.2. Usos d'OCL

El llenguatge OCL admet nombrosos usos en el context d'MDE:

a) Com a llenguatge de consulta sobre models.

b) Per a especificar invariants i restriccions d'integritat sobre les classes i els tipus d'un model de classes UML.

c) Per a especificar les precondicions i postcondicions de les operacions.

d) Per a descriure guardes en les màquines d'estats.

e) Per a especificar conjunts de destinataris dels missatges i accions UML.

f) Per a especificar restriccions en operacions.

g) Per a especificar regles de derivació d'atributs i cossos de les operacions.

En les assignatures de l'àrea d'enginyeria del programari del grau d'Enginyeria Informàtica es va introduir el llenguatge OCL juntament amb els seus tipus i operacions bàsiques. En aquest apartat presentarem el llenguatge des del punt de vista del seu ús en el context d'MDE, i introduirem algunes funcions una mica més avançades com l'especificació de la semàntica d'operacions, les restriccions d'integritat sobre els models i la creació de nous atributs i operacions.

Web recomanat

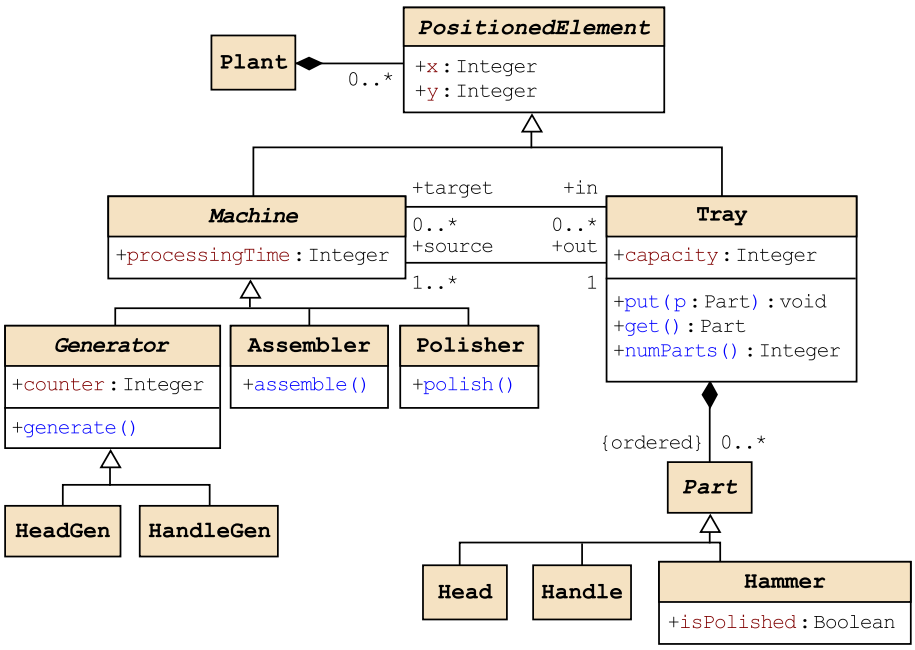
Tota la informació sobre OCL 2.3 es pot consultar en l'especificació de l'OMG: www.omg.org/spec/OCL/2.3.1

Vegeu també

Vegeu el mòdul "Documentació de requisits" de l'assignatura *Enginyeria de requisits* del grau d'Enginyeria Informàtica.

Per a il·lustrar l'ús d'OCL utilitzarem el metamodel de la cadena de muntatge que ja hem presentat en l'apartat 1 (figura 4), i que hem mostrat en la figura 10 enriquit amb algunes operacions en les classes per mostrar també com se n'especifica el comportament amb OCL.

Figura 10. El metamodel d'una cadena de muntatge



2.3. Restriccions d'integritat en OCL

Exemple

Exemples d'aquest tipus de restriccions són les multiplicitats dels extrems d'una associació UML.

Les restriccions d'integritat (o invariants) sobre un metamodel expressen les condicions que han de complir els models que representin sistemes vàlids. En la figura 10 podem veure, per la multiplicitat de l'associació, que una màquina de qualsevol model que representi una cadena de muntatge ha de tenir exactament una safata de sortida (`out`). Igualment, tota safata ha de tenir almenys una màquina que hi dipositi parts (`source`). Aquesta restricció, que s'expressa en termes de la multiplicitat dels extrems d'una associació, també pot ser expressada en OCL usant **invariants**:

```
context Tray inv: self.source->size() > 0
context Machine inv: self.out->size() = 1
```

Nota

Òbviament, tenint la possibilitat d'expressar multiplicitats directament en els diagrames de classes, no té gaire sentit utilitzar restriccions OCL com aquestes, encara que, com sempre, és decisió del modelitzador, i a nosaltres ens serveix per a il·lustrar-ne la naturalesa.

Recordem que el **context** d'una expressió OCL indica un tipus del model a les instàncies del qual fa referència l'expressió. Dins d'un context, la paraula reservada `self` s'usa per a referir-se a una instància d'aquest tipus. L'operació `size()` sobre col·leccions (`Set`, `Sequence`, `OrderedSet`, `Bag`) en permet conèixer la mida. El punt `(".")` i la fletxa `("->")` permeten navegar pels atributs i associacions dels elements, depenent de si són elements individuals o col·leccions, respectivament.

Com hem dit anteriorment, hi ha, no obstant això, altres restriccions que UML no és capaç d'expressar directament amb la seva notació. Per exemple, l'expressió OCL següent indica que el comptador de qualsevol generador ha de ser positiu:

```
context Generator inv PositiveCounter: self.counter >= 0
```

Noteu com és possible assignar noms als invariants (en aquest cas `PositiveCounter`) i a la resta de condicions OCL, per a poder referir-nos-hi posteriorment.

Una altra restricció interessant en el nostre exemple és la següent, anomenada `NoOverflow`, que imposa que les safates no poden contenir més peces de les indicades pel seu atribut `capacity`:

```
context Tray inv NoOverflow: self.part->size() <= self.capacity
```

Igual que en UML, en OCL és possible navegar per associacions amb extrems sense nom utilitzant el nom de la classe a l'altre extrem de l'associació, en minúscula (en aquest cas, `self.part`).

Vegeu també

Podeu trobar més informació sobre navegació en col·leccions en l'apartat "Especificació de la semàntica d'operacions".

Els invariants també ens serveixen per a definir apropiadament les característiques dels tipus del model.

Per exemple, en la figura 10 s'indica que hi ha tres tipus de màquines, però no queda reflectit que cadascuna pot imposar una restricció diferent sobre el nombre de safates a les quals pot estar connectada: polidores i acobladores haurien de tenir-ne almenys una safata d'entrada (com veurem a continuació, les acobladores en tindrien dues si no permetem safates amb elements de diferent tipus), mentre que les generadores no n'haurien de tenir cap. De la mateixa manera, podríem permetre que cada màquina tingués diverses safates de

sortida. No obstant això, atès que el nombre de safates de sortida està limitat a un, exigirem que les generadores no tinguin safates d'entrada, que les acobladores en tinguin exactament dues i que les polidores en tinguin exactament una.

Això es pot especificar mitjançant els tres invariants següents:

```
context Generator inv NoInputTrays:  self.in->isEmpty()
context Assembler inv TwoInTrays:  self.in->size() = 2
context Polisher inv OneInTray:  self.in->size() = 1
```

OCL admet nombroses maneres d'expressar invariants. Per exemple, els tres anteriors es poden especificar en un de sol, sobre la classe `Machine`, com segueix:

```
context Machine inv MachinesAndTrays:
   self.oclIsTypeOf(Generator) implies self.in->isEmpty() and
   self.oclIsTypeOf(Assembler) implies self.in->size() = 2 and
   self.oclIsTypeOf(Polisher) implies self.in->size() = 1
```

També és possible fer-ho usant una estructura condicional:

```
context Machine inv MachinesAndTrays2:
   self.in->size() = if self.oclIsTypeOf(Assembler) then 2
                     else if self.oclIsTypeOf(Polisher) then 1
                          else 0
                          endif
                     endif
```

OCL distingeix entre les operacions `oclIsTypeOf(T)` i `oclIsKindOf(T)`. La diferència és subtil però important: la primera determina si la classe a la qual pertany un objecte (és a dir, el seu classificador) és `T`, mentre que la segona determina si la classe a la qual pertany un objecte és `T`, o qualsevol superclasse de `T`.

Per exemple, si `g` és un objecte de classe `HeadGen`, llavors són certes les expressions següents: `g.oclIsTypeOf(HeadGen)`, `g.oclIsKindOf(HeadGen)`, `g.oclIsKindOf(Generator)`, `g.oclIsKindOf(Machine)` i `g.oclIsKindOf(PositionedElement)`. No obstant això, no ho són `g.oclIsTypeOf(Generator)` ni `g.oclIsTypeOf(Machine)`.

Si establim el nombre de safates d'entrada de cadascuna de les màquines ens hem d'assegurar que continguin parts dels tipus apropiats. Podem tenir diverses màquines col·locant peces en una mateixa safata, però totes les parts d'una mateixa safata haurien de ser del mateix tipus, la qual cosa es podria expressar com segueix:

```
context Assembler inv AllInGenerators:
   self.in->forAll(t | t.source->forAll(m | m.oclIsTypeOf(Generator)))
```

No obstant això, una restricció d'aquest tipus no seria suficient, ja que no podem permetre que, per exemple, una polidora tingui una safata de mànecs com a entrada. Necessitem que totes les màquines que posen peces en la safata d'entrada d'una acobladora siguin o generadores de mànecs o caps de martell, i a més tenir-ne de tots dos tipus:

```
context Assembler inv HeadAndHandleGens:
   self.in->forAll(t | t.source->forAll(m | m.oclIsTypeOf(Generator))
                and t.source->exists(m | m.oclIsTypeOf(HandleGen))
                and t.source->exists(m | m.oclIsTypeOf(HeadGen)))
```

Atès que hem fixat en dues el nombre de safates d'entrada a una màquina acobladora i que totes les parts d'una mateixa safata han de ser del mateix tipus, l'invariant `HeadAndHandleGens` es podria simplificar:

Exercici proposat

Hem restringit el tipus de màquines d'entrada a altres màquines, o per ser més precisos, les màquines que poden col·locar parts en les safates d'entrada de cada màquina. És necessari també establir restriccions sobre les de sortida?

```
context Assembler inv HeadAndHandleGens2:
   self.in->forAll(t | t.source->exists(m | m.oclIsTypeOf(HandleGen))
               and t.source->exists(m | m.oclIsTypeOf(HeadGen)))
```

De manera similar, una polidora només pot tenir acobladores com a màquines que posen parts en la seva safata d'entrada:

```
context Polisher inv :
   self.in->forAll(t | t.source->forAll(m | m.oclIsTypeOf(Assembler)))
```

Amb les restriccions establertes fins ara podem tenir situacions en què ens trobem amb diverses safates finals, de les quals cap màquina agafa peces. Podem imposar que hi hagi una única safata final, on es dipositaran totes les peces produïdes, amb l'invariant següent:

```
context Tray inv OneFinalTray:
   Tray.allInstances()->one(target->isEmpty())
```

En aquest invariant hem fet ús de l'operació `allInstances()` que, aplicada a un nom d'un tipus, retorna el conjunt d'instàncies que actualment hi ha en el model d'aquest tipus. Per la seva banda, el quantificador `one()`, aplicat a una col·lecció, retorna `true` si hi ha exactament un únic element que compleix una propietat. OCL també ofereix la funció `any()`, que retorna algun dels elements d'un conjunt que satisfà una propietat. Si hi ha més d'un que la satisfà, en pot retornar qualsevol. Si no n'hi hagués cap, retornaria el valor `null`.

Exercici proposat

Suposem que eliminem l'invariant `OneFinalTray`. Poseu un exemple de sistema que compleixi amb la resta dels invariants i que tingui més d'una safata final. Discutiu la necessitat d'afegir l'invariant `OneFinalTray`.

En OCL cal distingir entre els valors `null` i `invalid`. El primer és un valor que indica "absència de valor". Per exemple, un extrem d'una associació amb multiplicitat `0..1` que no està associat a cap classe pren el valor `null`. D'altra banda, `invalid` és un valor que indica que l'operació que es tracta d'executar és incorrecta, com per exemple quan es fa una divisió per zero. És important destacar que `null` és un valor vàlid que pot ser emmagatzemat en col·leccions, cosa que no ocorre amb `invalid`. El que no poden és executar-se operacions sobre `null`, excepte les corresponents a `Bag` (per exemple, `null->isEmpty()` retorna `true`, i `null->notEmpty()` retorna `false`); no obstant això, `isOclType(T)` retorna `invalid` si `T` és `null`. OCL proporciona les operacions `oclIsInvalid()` per saber si un element és `invalid`, i `oclIsUndefined()` per a saber si és `null` o `invalid`.

L'operació `allInstances()` és molt útil quan volem imposar una restricció sobre el nombre d'objectes que hi pot haver en un model. Per exemple, podem voler indicar que el nombre de màquines d'un sistema ha de coincidir amb el nombre de safates:

```
context Plant inv SameNumbers:
   Machine.allInstances()->size() = Tray.allInstances()->size()
```

Noteu que aquest invariant és potser més restrictiu del necessari. Encara que és per descomptat decisió del modelitzador afegir-lo o no, en principi no hi ha problema perquè diverses màquines utilitzin una mateixa safata com a safata de sortida.

Exercici proposat

L'invariant `SameNumbers` limita dràsticament les configuracions de màquines i safates permeses; podeu posar dos exemples de configuracions amb diferent nombre de màquines?

La restricció `NoOverflow2` és equivalent a la restricció `NoOverflow` que hem vist anteriorment:

```
context Tray inv NoOverflow2:
   Tray.allInstances()->forAll(t | t.part->size() <= t.capacity)
```

És important també establir que no hi hagi dues màquines o safates en la planta amb la mateixa posició:

```
context PositionedElement inv NoOverlap:
   PositionedElement.allInstances()->forAll(m1,m2 |
         m1<>m2 implies (m1.x<>m2.x  or  m1.y<>m2.y) )
```

Una situació molt comuna en els diagrames UML és que poden permetre alguns cicles que no s'han de donar en els sistemes que modelitzen. Per exemple, sense considerar les restriccions establertes fins ara, el diagrama UML de la figura 10 admet cicles que permeten a una màquina estar connectada a una mateixa safata com a safata d'entrada i de sortida. Per evitar aquesta situació podem imposar l'invariant següent:

```
context Machine inv NoCycles:
   self.out->forAll(t | t.target->excludes(self)) and
   self.in->forAll(t | t.source->excludes(self))
```

Exercici proposat

És realment l'invariant `NoCycles` deduïble de la resta dels invariants? Justifiqueu la resposta.

Com és habitual, aquest invariant admet diverses formulacions equivalents, com per exemple:

```
context Machine inv NoCycles2:
   self.out->intersection(self.in)->isEmpty()
```

Normalment, el millor context és aquell amb el qual l'invariant es pot expressar de la manera més simple, o bé amb el qual l'invariant és més simple de comprovar; quant a la completesa de les nostres especificacions, serà en última instància responsabilitat del modelitzador, i per a això l'experiència d'aquest serà fonamental.

Lectura recomanada

J. Cabot; E. Teniente (2007). "Transformation techniques for OCL constraints". *Science of Computer Programming* (vol. 68, núm. 3, pàg. 179-195).

2.4. Creació de variables i operacions addicionals

Una cosa que també permet OCL és definir nous atributs i operacions en el model, usant la paraula reservada `def`. L'objectiu d'aquestes operacions i atributs serà normalment simplificar les expressions OCL necessàries. Les dues expressions següents declaren, respectivament, un nou atribut (`isFinal`) per a la classe `Tray`, i una operació per a la classe `PositionedElement`. L'atribut determina si una safata és final o no, i l'operació calcula la distància de Manhattan des d'un altre objecte d'aquesta classe fins a aquell.

```
context Tray
   def: isFinal: Boolean = self.target->isEmpty()
context PositionedElement
   def: distanceTo(p : PositionedElement) : Integer =
            (p.x - self.x).abs() + (p.y - self.y).abs()
```

OCL requereix que els nous atributs siguin sempre derivats, i que les operacions definides així siguin només de consulta.

2.5. Assignació de valors inicials

Una cosa fonamental per als nostres diagrames UML és tenir la possibilitat d'afegir regles que assignin valors inicials als atributs i als extrems de les associacions.

Les regles que expressen valors inicials són molt simples, ja que n'hi ha prou de detallar el context, l'atribut i el valor inicial que volem que prengui en el moment en el qual es crea un objecte d'aquest tipus. En el nostre exemple, podem suposar que inicialment les màquines generadores parteixen de 100 peces cadascuna, i que els martells quan es creen no estan polits:

```
context Generator::counter : Integer
   init = 100
context Hammer::isPolished : Boolean
   init = false
```

OCL també permet expressar els valors dels atributs derivats, amb la clàusula `derive`. El diagrama de la figura 10 no conté atributs derivats, però suposem que la classe `Tray` en té un, denominat `connectedMachines`, que conté el nombre de màquines d'entrada i de sortida que a cada moment té connectada una safata. El valor d'aquest atribut derivat s'especificaria en OCL de la manera següent:

```
context Tray::connectedMachines : Integer
   derive: self.source->size() + self.target->size()
```

2.6. Col·leccions

Vegeu també

Les col·leccions s'introdueixen en l'assignatura *Enginyeria de requisits* del grau d'Enginyeria Informàtica.

Quan es treballa amb col·leccions d'objectes en OCL, cal tenir en compte la diferència entre les quatre col·leccions disponibles. En un `Set`, cada element només pot aparèixer una vegada. En un `Bag`, els elements poden aparèixer més d'una vegada. Un `Sequence` és un `Bag` en què els elements estan ordenats. Un `OrderedSet` és un `Set` en què els elements estan ordenats.

Vegeu també

Vegeu la descripció de les operacions bàsiques en el mòdul "Documentació de requisits" de l'assignatura *Enginyeria de requisits*.

Cadascuna d'aquestes col·leccions admet una sèrie d'operacions bàsiques per a manejar-les (`size()`, `select()`, `collect()`, `sum()`, etc.). No obstant això, hi ha alguns aspectes menys coneguts del comportament de les col·leccions, sobretot pel que fa a com es construeixen quan es navega a través de les associacions d'un diagrama de classes d'UML o de MOF, i que val la pena presentar aquí.

2.6.1. Navegacions que resulten en `Sets` i `Bags`

En OCL hi ha la regla que quan es navega a través de més d'una associació amb multiplicitat més gran que `1`, s'obté un `Bag`.

> Per exemple, quan es va des d'una instància de classe `A` a través de més d'una instància de classe `B` fins a més d'una instància de classe `C`, el resultat és un `Bag` d'objectes de tipus `C`. No obstant això, quan es navega només una d'aquestes associacions, el resultat de la navegació és un `Set`.

Per poder entendre per què és important aquest fet, suposem que volem conèixer quantes màquines estan connectades a una màquina determinada a través de les seves safates, i per a això creem l'expressió OCL següent:

```
context Machine def: neighbours : Integer =
   self.in.source->union(self.out.target)->size()
```

Aquesta expressió planteja, no obstant això, un problema. Una màquina pot estar connectada a més d'una safata d'entrada, i per tant es podria repetir una referència al mateix objecte de la classe `Machine` en aquestes col·leccions, ja que són `Bags` i no `Sets`. Així, en l'expressió escrita anteriorment, una màquina pot ser comptada dues vegades, la qual cosa no seria correcte. Per a això, OCL ofereix operacions que transformen un dels tipus de col·leccions en qualsevol dels altres, com per exemple `asSet()` o `asBag()`. En aquest cas, usant una d'aquestes operacions corregim la definició anterior:

```
context Machine def: neighbours : Integer =
   self.in.source->union(self.out.target)->asSet()->size()
```

2.6.2. Navegacions que resulten en `OrderedSets` i `Sequences`

Quan es navega una associació marcada com a `ordered`, la col·lecció resultant és de tipus `OrderedSet`. Així mateix, quan es navega a través de més d'una associació, i una és marcada com a `ordered`, la col·lecció resultant és de tipus `Sequence`. Moltes operacions estàndard tenen en compte l'ordre de la col·lecció, com per exemple: `first()`, `last()` o `at()`.

2.7. Especificació de la semàntica de les operacions

UML permet especificar de diferents maneres el comportament d'un sistema. Una de les més usuals és mitjançant l'especificació d'operacions associades a les classes, les quals serveixen per a descriure el comportament dels objectes del sistema. OCL permet especificar la semàntica de les operacions en termes de les seves precondicions i postcondicions.

Lectura recomanada

L'ús de precondicions i postcondicions i invariants per a l'especificació d'operacions de programari ha estat comú en informàtica per molt temps, encara que va ser popularitzat amb el que Bertrand Meyer denomina *disseny per contracte*. Vegeu: **B. Meyer** (1997). *Object-Oriented Software Construction* (2a. ed.). Prentice Hall.

El propòsit d'una precondició és indicar les restriccions que s'han de satisfer perquè l'operació es pugui executar correctament. El propòsit d'una postcondició és especificar les condicions que s'han de satisfer després de l'execució de l'operació (suposada la seva precondició en començar l'execució). En altres paraules, les precondicions representen els estats vàlids del sistema per als quals està garantit que l'operació es pot executar correctament, i si ho fa ha d'acabar en un estat en el qual se satisfaci la postcondició.

Noteu que les precondicions i postcondicions determinen **què** ha de fer una operació, però no indiquen **com** ho ha de fer. D'una altra manera estaríem detallant la implementació concreta del sistema, la qual cosa no és desitjable per a un model, i menys encara per a un model independent de la plataforma (PIM).

> Per exemple, la postcondició d'una operació que ordena un vector d'enters ha de determinar que el vector ha d'estar ordenat després de l'execució de l'operació, però no ha d'indicar si l'ordenació es fa utilitzant *quicksort*, inserció o qualsevol altre mètode particular d'ordenació.

En OCL les precondicions s'indiquen amb la paraula reservada `pre` i les postcondicions amb `post`. La paraula reservada `result` representa la variable especial que emmagatzema el resultat d'una operació que retorna un valor. L'expressió OCL següent especifica el comportament de l'operació `numParts()` que retorna el nombre de peces d'una safata, descrivint el que ha de retornar després de l'execució.

```
context Tray::numParts() : Integer
post: result = self.part->size()
```

OCL també permet una manera alternativa de modelitzar consultes simplement especificant el valor de la consulta com el cos de l'operació, amb la clàusula `body`:

```
context Tray::numParts() : Integer
body: self.part->size()
```

L'absència d'una precondició (com és el cas en les expressions anteriors) equival a indicar que qualsevol estat inicial és vàlid per a l'operació. De la mateixa manera, és possible especificar més d'una precondició o postcondició per a una mateixa operació, i en aquest cas totes s'han de complir.

En una postcondició, a més de la variable especial `result`, es pot usar el sufix `@pre` per a referir-se al valor d'una variable just abans de l'execució de l'operació, és a dir, el valor que tenia en avaluar-se la precondició. També hi ha l'operació `oclIsNew()` per a consultar si un objecte determinat no existia abans de l'execució de l'operació; en altres paraules, és nou i l'ha creat l'operació. El sufix `@pre` pot ser útil, per exemple, per a especificar el comportament de l'operació `get()` de les safates:

```
context Tray::get() : Part
pre:  self.part->notEmpty()
post: result = self.part@pre->first() and self.part->excludes(result)
```

Web recomanat

OCL defineix nombroses operacions sobre col·leccions (`first()`, `excludes()`, etc.). La definició de cadascuna es pot consultar en l'estàndard d'OCL mateix: www.omg.org/spec/OCL/2.3.1

En aquest cas hem demanat que la peça que obtinguem sigui la primera de la safata. També seria possible indicar que en podem obtenir qualsevol:

```
context Tray::get() : Part      --- alternative specification
pre:  self.part->notEmpty()
post: self.part@pre->size() = self.part->size() + 1
      and self.part@pre->includes(result)
      and self.part->excludes(result)
```

De manera similar, una possible especificació OCL de l'operació `put()` és la següent:

```
context Tray::put(p : Part)
pre:  self.part->size() < self.capacity and self.part->excludes(p)
post: self.part = self.part@pre->append(p)
```

Aquesta especificació fa ús de l'operació `append()` sobre col·leccions ordenades. En el cas que vulguem no fixar que l'operació col·loqui la peça al final de la safata, l'especificació es podria fer de la manera següent:

```
context Tray::put(p:Part)     --- alternative specification
pre:  self.part->size() < self.capacity and self.part->excludes(p)
post: self.part->size() = self.part@pre->size() + 1
      and self.part->includes(p)
```

És important comentar que una precondició o postcondició pot estar especificada de manera incompleta. Per exemple, les postcondicions anteriors de l'operació `put()` no indiquen explícitament que la resta del `Tray` no canvia.

2.8. Invocació d'operacions i senyals

OCL disposa d'un operador denominat `hasSent` (abreujat "`^`") per a indicar que s'ha dut a terme una comunicació entre dos objectes. Per *comunicació OCL* entén l'emissió d'un senyal, o la invocació d'una operació. Aquest operador només es pot usar en les postcondicions de les operacions.

Per exemple, l'expressió `self.out^put(h)` afirma que durant l'execució de l'operació `genHandle()` del generador de mànecs s'ha hagut d'invocar l'operació `put()` de la seva safata de sortida amb el mànec `h`. En altre cas, l'expressió és `false`. Per exemple:

```
context       HandleGen::generate1()
post: let h : Handle = … in self.out^put(h)
```

La clàusula `let...in...` permet definir variables que es poden usar diverses vegades en una mateixa expressió OCL.

També és possible deixar sense especificar el valor concret dels paràmetres d'una invocació, indicant que solament ens importa comprovar que s'ha invocat l'operació, independentment dels paràmetres que s'hagin usat. Això es fa usant un signe d'interrogació (`"?"`) com a nom del paràmetre:

```
context HandleGen::generate2()
post: … self.out^put(? : Handle)  …
```

OCL defineix un tipus denominat `OclMessage`, al qual pertanyen tots els missatges usats en l'emissió de senyals i la invocació d'operacions. També es defineix l'operador `SentMessages` (abreujat "^^"), que retorna el conjunt de missatges enviats. Per exemple, en la postcondició de l'operació `generate2()` anterior es podria indicar que s'ha invocat només un missatge `put()`, com segueix:

```
self.out^^put(? : Part)->size() = 1
```

Podem completar les especificacions de les operacions `generate()`, de `HandleGen` i `HeadGen` proporcionant les precondicions adequades i assegurant-nos en les postcondicions que les peces retornades són objectes nous.

```
context HeadGen::generate()
pre:  self.counter > 0 and self.out.numParts() < self.out.capacity
post: let msg : OclMessage = self.out^put(? : Head)  in
        counter = counter@pre - 1  and  msg.p.isOclNew()

context HandleGen::generate()
pre:  self.counter > 0 and self.out.numParts() < self.out.capacity
post: let msg : OclMessage = self.out^put(? : Handle)  in
        counter = counter@pre - 1  and  msg.p.isOclNew()
```

Noteu com s'accedeix al paràmetre del missatge usant el nom definit per a aquest en l'operació (en aquest cas el paràmetre de `put()` es denominava `p`, i per tant hi accedim com a `msg.p`).

Igualment, podríem accedir a les seqüències de missatges `get()` i `put()` usats per la màquina acobladora en la seva operació `assemble()` i comprovar que s'han emès exactament dos missatges `get()` i un `put()`, i que el paràmetre del `put()` és precisament de tipus `Hammer`:

```
context Assembler::assemble()
pre: -- hi ha peces a les 2 safates d'entrada
      self.in->forAll(numParts()>0)
      and -- la safata de sortida no és plena
      self.out.numParts() < self.out.capacity
post: let t1 : Tray =self.in->asSequence()->first() in  -- una safata
      let t2 : Tray =self.in->asSequence()->last() in-- una altra safata
      let msg : OclMessage = self.out^put(? : Part) in -- emès un put()
      let h : Hammer = msg.p in        -- el arg. del put és un martell
         self.in^^get()->size() = 2    -- se n'han emès dos get()
     and t1^get()  and  t2^get()     -- un get()amb cada safata
     and m1.hasReturned()         -- tots dos van tornar correctament
     and m2.hasReturned()
     and self.out^^put(? : Part)->size() = 1 -- se n'ha emès un de sol put
     and h.isOclNew()                -- el martell del put és nou
     and not h.isPolished            -- i no estava polit
```

Noteu l'ús en aquesta expressió de l'operador `^^` per a referir-se a tots els missatges d'un tipus enviats o rebuts a un determinat tipus d'objectes. Noteu també com s'accedeix al paràmetre del missatge (`msg.p`) usant el nom definit per a aquest en l'operació (`p`). Finalment, noteu l'ús de les operacions `first()` i `last()`. Sobre estructures ordenades (`Sequence` i `OrderedSet`), aquestes operacions proporcionen, respectivament, el primer i l'últim dels seus elements. Atès que les safates d'entrada d'una màquina són un conjunt no ordenat, hem d'aplicar abans l'operació `asSequence()` (o `asOrderedSet()` depenent del cas) per obtenir l'element que volem. Atès que per altres restriccions OCL establertes sabem que una màquina acobladora té exactament dues màquines generadores com a màquines col·locant elements en les seves safates d'entrada, en tenim prou d'agafar-ne una i una altra i estarà assegurat que siguin d'elements diferents.

Amb els operadors `^` i `^^` sabem com referir-nos als senyals emesos i operacions invocades durant l'execució d'una operació. Els senyals són comunicacions unidireccionals, i no retornen cap valor; no obstant això, les operacions impliquen una invocació i un retorn (o terminació), que conté el valor retornat per l'operació.

OCL ofereix dues funcions per a gestionar el retorn d'una operació i el valor retornat per aquesta: `hasReturned()` i `result()`. Totes dues pertanyen a la classe `OclMessage`. Per descomptat, l'operació `result()` només retorna un valor ben definit si l'operació cridada ha acabat i ha retornat un valor. Això és precisament el que es comprova amb `hasReturned()`. Si l'operació no ha acabat, `hasReturned()` retorna `false` i `result()` està indefinit (retorna `invalid`).

Com a part de l'especificació de la màquina polidora, usant aquestes funcions podem especificar que s'ha d'haver enviat un missatge `get()` a la safata d'entrada, que aquesta ha retornat un martell i que aquest ha estat enviat en un missatge `put()`, després de polir-lo, a la safata de sortida. L'expressió OCL següent especifica aquesta operació:

```
context Polisher::polish()
pre:  (self.in.numParts()>0) and (self.out.numParts()<self.out.capacity)
post: let   inM : OclMessage = self.in^get() in
      let   h: Hammer= inM.result() in -- el get()ens dóna un martell
         inM.hasReturned()    -- missatge enviat i resposta rebuda
         and h.isPolished        -- el martell ha estat polit i està llest
         and self.out^put(h)  -- l'objecte tornat per get() és el
                  -- mateix que després s'envia a la safata de sortida
```

2.9. Alguns consells de modelització amb OCL

En general, modelitzar un sistema no és una tasca simple. A part de la complexitat intrínseca que pugui tenir l'especificació del sistema mateix, també ens trobem amb la dificultat "accidental" que ocasiona l'ús de notacions com UML o OCL. Per exemple, en aquest capítol hem vist com una mateixa condició sobre un model es pot especificar amb diverses expressions OCL equivalents (des de contextos diferents, usant `self` o `allInstances()`, per exemple). Els apartats següents contenen algunes recomanacions sobre l'ús d'UML i OCL per a l'especificació de sistemes.

2.9.1. Ús de la representació gràfica d'UML enfront d'OCL

Hem vist que hi ha moltes restriccions que admeten dues representacions: una de manera gràfica en UML i una altra com a invariants en OCL. Els exemples més típics d'aquest tipus de restriccions són les multiplicitats dels extrems de les associacions.

R1: expressar les restriccions de manera gràfica sempre que sigui possible.

En altres paraules, és millor utilitzar OCL només per a les expressions que no es puguin expressar de cap manera amb la notació gràfica d'UML.

2.9.2. Consistència dels models

Quan estem definint els models és possible que introduïm massa restriccions, o que les que especifiquem siguin incompatibles entre si. Això és una cosa que hem de tenir molt en compte.

R2: assegurar que el model proposat és consistent.

Una bona pràctica és comprovar, sempre que introduïm una nova restricció, que hi ha almenys una instància del model que pot satisfer totes les restriccions d'integritat al mateix temps.

2.9.3. Compleció dels diagrames UML

És important incloure en els diagrames UML tota la informació dels elements d'un model, i especialment les associacions:

R3: especificar de manera explícita els noms de tots els extrems de les associacions entre les classes, i també les seves multiplicitats.

En UML, quan hi ha més d'una associació entre dues classes és obligatori assignar noms als extrems de les associacions per a poder distingir-los. No obstant això, quan només hi ha una associació entre dues classes, UML no obliga a fer-ho. En aquest cas, quan OCL necessita referir-se als objectes que estan relacionats amb altres per mitjà d'una associació sense nom, utilitza la convenció d'usar com a nom de l'extrem de l'associació el nom de la classe però en minúscula. Això ho hem il·lustrat ometent el nom dels extrems de l'associació entre les classes `Tray` i `Part`. De totes maneres, és convenient indicar el nom de tots els extrems de les associacions entre les classes.

Nota

Recordeu que el nom d'un extrem d'associació representa el paper que tenen en l'associació els objectes d'aquesta classe, i se sol expressar amb un nom (i no amb un verb). Des del punt de vista d'OCL, aquest nom es converteix a més en el nom d'un atribut de la classe de l'extrem oposat, per la qual cosa ha de tenir sentit com a nom d'atribut.

Igualment, és important especificar de manera explícita la multiplicitat de tota associació. Per defecte, UML suposa que la multiplicitat d'un extrem d'associació és 1 tret que s'indiqui un altre valor. No obstant això, és un error comú pensar que la multiplicitat per defecte és "`*`". Per a evitar malentesos és preferible no deixar res sense detallar de manera explícita.

Per descomptat, és important detallar altres característiques importants de les associacions, com per exemple si estan ordenades (`isOrdered`), si no permeten que un mateix element aparegui dues vegades (`isUnique`), etc. És important des del punt de vista de la modelització que ens plantegem aquestes qüestions per a cada associació.

2.9.4. La importància de ser exhaustiu

A més de tenir molt en compte les propietats de les associacions i de les classes que conformen el model (quins són abstractes i quins no, etc.), també cal preveure totes les restriccions d'integritat dels models. En aquest capítol hem especificat diversos invariants sobre el model usant OCL. No obstant això, ens sorgeix el dubte de si n'hi haurà més que hàgim pogut oblidar, o no hàgim tingut en compte.

R4: identificar totes les restriccions que han de complir els models que han de representar models vàlids del sistema.

Aquesta no és, per descomptat, una tasca fàcil, i no hi ha tampoc receptes màgiques. Un dels problemes habituals és que el modelitzador se sol oblidar d'incloure restriccions que especifiquen situacions òbviament impossibles en la realitat, però que el model UML sí permet. És per això que és important tractar d'analitzar totes les associacions per si es poguessin donar cicles no desitjats, analitzar els valors dels atributs per si requerissin ser restringits, les del model complet per si calguessin restriccions d'unicitat (per exemple, que el valor d'alguns atributs sigui únic), els valors inicials dels atributs, etc.

Nota

Sol ser habitual oblidar-se d'imposar restriccions del tipus "dos elements d'una planta de muntatge no poden compartir una mateixa posició física".

2.9.5. Navegació complexa

A l'hora d'escriure expressions OCL solem "navegar" per les associacions del model per a referir-nos als elements que estan relacionats amb un de determinat.

En OCL seria possible escriure totes les expressions començant des d'un context qualsevol, però això podria produir expressions molt llargues i complexes d'entendre, ja que en alguns casos obligaria a incloure camins de navegació molt llargs per a referir-se a objectes relacionats de manera distant.

R5: cal tractar d'escollir el millor context per a cada expressió, i d'usar expressions de navegació com més senzilles millor.

A l'hora d'escollir el context d'un invariant, cal tenir en compte aquestes indicacions:

- Si l'invariant restringeix el valor d'un atribut d'una classe, aquesta classe constitueix el context òptim per a l'invariant.

- Si l'invariant restringeix el valor dels atributs de diverses classes, és millor escollir com a context la classe des d'on sigui més fàcil expressar la restricció OCL.

- És millor tractar d'usar expressions amb `self` en comptes de fer-ho amb `allInstances()` sempre que es pugui, per simplificar les expressions OCL.

- Utilitzar variables i operacions auxiliars (definides amb clàusules `def`) per a simplificar les expressions que resultin molt complicades. Aquest tipus de variables es poden veure com una generalització de l'expressió `let` per a variables que s'utilitzen en més d'una sola expressió OCL.

2.9.6. Modularització d'invariants i condicions

Sol ser habitual expressar certs invariants, precondicions i postcondicions amb moltes restriccions unides per clàusules `and`. En comptes d'això, i atesa la facilitat per a compondre condicions mitjançant conjunció que té OCL (n'hi ha prou d'incloure-les unes després d'altres), és millor descompondre les condicions en expressions més petites que es poden combinar després mitjançant conjuncions, i així modularitzar les especificacions.

R6: modularitzar els invariants i les condicions tant com sigui possible.

El fet de definir expressions més simples i manejables en millorarà la llegibilitat, comprensibilitat i manteniment notablement.

2.10. Conclusions

Tradicionalment, la modelització de sistemes de programari ha estat sinònim d'especificació de diagrames. La majoria dels models consistien en figures amb forma de caixes i fletxes, i amb algun text que les acompanyava. La informació recopilada per aquest tipus de models té la tendència a ser incompleta, imprecisa, informal i, fins i tot de vegades, inconsistent.

Per això cal disposar de llenguatges precisos i amb base formal per a especificar apropiadament els models. Això és fins i tot més important quan el model no ha de ser llegit per humans, sinó per màquines que han de dur a terme totes les missions que defineix MDE: generar automàticament el codi final de la implementació, les proves, comprovar la consistència del model, simular-lo, analitzar-ne les propietats, etc. El problema és que automatitzar aquest treball només és possible si el model conté tota la informació necessària. Una màquina no pot interpretar regles escrites en llenguatge humà, o fer suposicions per molt òbvies que ens semblin a nosaltres[3].

[(3)] Per exemple, que els comptadors dels generadors han de ser positius, o que dues màquines no poden estar situades en les mateixes coordenades en la planta

No obstant això, un model escrit en un llenguatge que usa expressions pot no ser fàcil d'escriure, entendre, modificar o mantenir. Per això la comunitat d'MDA ha optat per la combinació d'UML i OCL com a llenguatges per a especificar models de la manera més intuïtiva i precisa alhora. En aquest capítol hem presentat els principals conceptes i mecanismes del llenguatge OCL, usant exemples per a això, i hem donat algunes indicacions de com usar OCL de la millor manera possible. A continuació estudiarem com és possible també definir nous llenguatges de modelització. Això és necessari quan UML, en ser un llenguatge de propòsit molt general, no proporciona la notació adequada per a modelitzar els nostres sistemes concrets.

3. Llenguatges específics de domini

3.1. Introducció

En els capítols anteriors hem discutit la necessitat de descriure, de manera precisa, i a un nivell d'abstracció adequat, tots els aspectes d'un sistema que són rellevants des d'un punt de vista determinat. També hem destacat la importància d'utilitzar el llenguatge més adequat per a la descripció d'aquests aspectes, que tingui l'expressivitat més apropiada per a especificar-los. Per a això es poden usar tant llenguatges de modelització de propòsit general (per exemple UML) com també llenguatges específics de domini (o DSL, *domain-specific languages*).

Un **llenguatge específic de domini (DSL)** és un llenguatge que proporciona els conceptes, notacions i mecanismes propis d'un domini en qüestió, semblants als que manegen els experts d'aquest domini, i que permet expressar els models del sistema amb un nivell d'abstracció adequat.

Expert del domini

Un expert del domini és aquella persona que és especialista en el domini del problema, encara que no necessàriament en el domini de l'aplicació. Per exemple, un banquer coneix bé la terminologia que s'utilitza en la banca i un mariner la de la navegació, encara que cap d'ells pot no tenir els coneixements per a dissenyar una aplicació bancària o de navegació usant UML o implementar-la fent ús de tecnologies Java o .NET.

Hi pot haver DSL de naturalesa i grau de complexitat molt diferents, des de molt tècnics[4] a aquells de molt alt nivell orientats a persones de negoci o científics[5].

(4)Per exemple, els que permeten configurar una xarxa de commutació de paquets o especificar els paràmetres d'una línia de productes de programari.

Els DSL proporcionen els conceptes essencials d'un domini d'aplicació com a elements de primera classe d'aquest llenguatge.

(5)Per exemple, els que permeten modelitzar processos de negoci, cadenes de producció d'una fàbrica d'acoblament, molècules de DNA, etc.

Domain specific modeling

La proliferació dels DSL ha propiciat l'aparició del terme DSM (*domain specific modeling*) que es refereix a la disciplina dins d'MBE que usa DSL per a modelitzar dominis d'aplicació concrets.

Web recomanat

Vegeu http://www.dsmforum.org

D'aquesta manera, els DSL permetran als experts del domini expressar el model del seu sistema usant el vocabulari que utilitzen normalment, i de manera independent de les plataformes d'implementació.

En general, els DSL es dissenyen i desenvolupen perquè siguin els experts del domini els qui modelitzin els seus sistemes propis, i que després aquests models puguin ser processats de manera automàtica per ordinadors.

Això obliga a conjugar simplicitat amb precisió: els models han de ser fàcilment entesos i manejables pels experts del domini, però al seu torn han de ser prou precisos perquè siguin executables i processables pels sistemes d'informació.

La connexió entre els models expressats en un DSL i els sistemes en què s'executen i analitzen es fa mitjançant les transformacions de models, que són les encarregades de convertir els models en els artefactes de programari corresponents (codi font, programes, components, aplicacions, etc.) en les plataformes de programari destinació.

Vegeu també

Les transformacions de models s'estudien amb més detall en l'apartat 4.

Aquest capítol està dedicat a aquests llenguatges, les seves característiques i propietats, i també a les maneres existents actualment per a definir-los de manera que descriguin correctament els models dels nostres sistemes.

3.2. Components d'un DSL

Encara que, com hem esmentat abans, hi ha molt tipus de DSL depenent dels dominis per als quals són creats, tots comparteixen una sèrie de components comuns que passem a descriure.

3.2.1. Sintaxi abstracta

La **sintaxi abstracta** d'un llenguatge descriu el vocabulari amb els conceptes del llenguatge, les relacions entre ells, i les regles que permeten construir les sentències (programes, instruccions, expressions o models) vàlides del llenguatge.

Aquestes regles restringeixen els possibles elements dels models vàlids, i també aquelles combinacions entre elements que respecten les regles semàntiques del domini.

Com hem comentat en l'apartat 1, els **metamodels** constitueixen la manera més natural de descriure la sintaxi abstracta d'un llenguatge.

El metamodel de la figura 8 del primer capítol representava els conceptes que es manegen a l'hora de dissenyar una cadena de muntatge d'una fàbrica de martells i les relacions que s'hi poden establir per a formar models vàlids. En l'apartat 2 hem vist com també cal incloure restriccions OCL per a especificar algunes regles que no es poden descriure amb una notació gràfica com la que proporciona UML.

És important assenyalar que la sintaxi abstracta és independent de la **notació** que s'usi per a representar els models (sigui textual o gràfica) i de la semàntica, o **interpretació**, que els vulguem donar, tant a escala de l'estructura del sistema com del comportament.

3.2.2. Sintaxi concreta

Tots els llenguatges (no solament de modelització o de programació) tenen una notació que en permet la representació i construcció dels models.

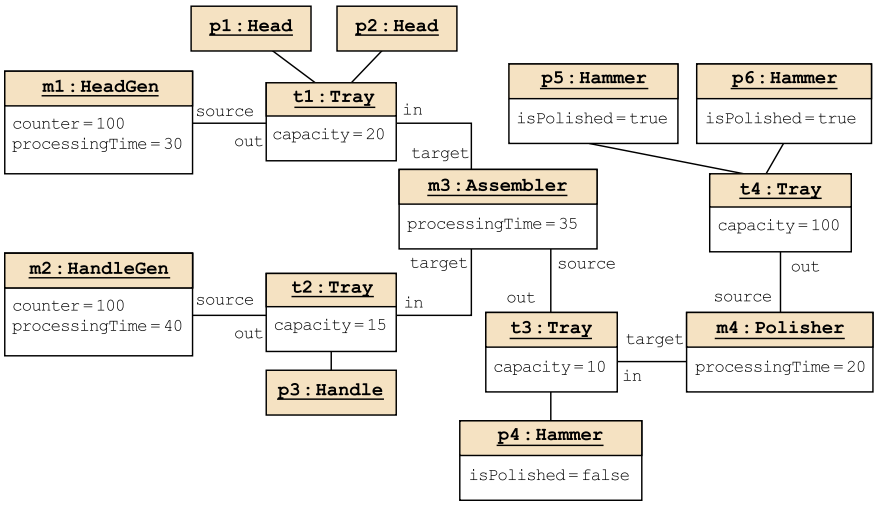
La **sintaxi concreta** d'un llenguatge defineix la notació que s'usa per a representar els models que es poden descriure amb aquest llenguatge.

Principalment hi ha dos tipus de sintaxis concretes: textuals i gràfiques.

La **sintaxi gràfica** (o **visual**) descriu els models de manera diagramàtica, usant símbols per a representar els elements i les relacions entre aquests.

Per exemple, en UML s'utilitzen caixes rectangulars per a representar classes i instàncies, i línies per a representar associacions i enllaços. Així, les figures 7, 8 i 9 del primer capítol representen visualment un model i el seu metamodel (recordem que un metamodel és també un model). La figura 9 mostra la representació de la figura 8 usant la sintaxi concreta d'UML.

Figura 11. El model de la cadena de muntatge de martells de la figura 9 en UML



L'avantatge principal d'una sintaxi gràfica és que permet representar molta informació de manera intuïtiva i fàcilment comprensible. Un dels inconvenients principals és que no permet especificar els sistemes amb massa nivell de detall, ja que els diagrames es compliquen excessivament. A més, l'expressivitat de les notacions gràfiques sol ser limitada; per això se solen complementar amb especificacions descrites en notacions textuals, com hem vist en l'apartat anterior amb UML i OCL.

La **sintaxi textual** permet descriure models usant sentències compostes per cadenes de caràcters, d'una manera similar a com fan la majoria dels llenguatges de programació.

El model de la figura 11 representat usant la notificació HUTN

Un exemple de notació textual és la que usa OCL, que hem vist en l'apartat anterior. XMI és una altra notació textual, que usa XML per a representar models. OMG també ha definit una altra notació textual per a descriure models i metamodels, denominada *HUTN* (*human-usable textual notation*), que no usa XML per com és de carregada i complexa aquesta notació per a ser entesa i utilitzada per humans. Usant HUTN, el model de la figura 11 es representa com segueix:

```
HeadGen "m1" { counter: 100 processingTime: 30 out: "t1" }
HandleGen "m2" { counter: 100 processingTime: 40 out: "t2" }
Assembler "m3" { processingTime: 35 in: "t1", "t2" out: "t3" }
Polisher "m4" { processingTime: 20 in: "t3" out : "t4" }
Tray "t1" { capacity: 20 source: "m1" target: "m3"
          part: Handle "p1" {}, Handle "p2" {} }
Tray "t2" { capacity: 15 source: "m2" target: "m3" part: Handle "p3" {}
Tray "t3" { capacity : 10 source : "m3" target: "m4"
          part: ~isPolished Hammer "p4" {} }
Tray "t4" { capacity : 100 source : "m4"
          part: isPolished Hammer "p5" {}, isPolished Hammer "p6" {} }
```

Noteu com es tracta de facilitar en HUTN la comprensió per part dels usuaris (humans) de les especificacions, mitjançant alguns mecanismes concrets. Per exemple, els noms dels objectes apareixen darrere dels seus tipus; o els atributs booleans es poden mostrar com a adjectius dels objectes, davant d'aquests. Així, en el codi font que apareix en el text de dalt s'usa `isPolished Hammer"p5"` per a indicar que l'atribut `isPolished` d'aquest martell pren el valor `true`, mentre que `isPolished Hammer "p4"` indica que el valor de `isPolished` és `false` en l'objecte el nom del qual és `p4`.

Consulta recomanada

OMG (2004). *Human-Usable Textual Notation (HUTN) Specification* (v1.0). *OMG doc. formal* (04-08-01).

Exercici proposat

Tracteu de llegir i comprendre la descripció XMI que genera qualsevol eina UML del model representat en la figura 11, i compareu-la amb la seva representació en HUTN mostrat en el codi font que apareix en el text de dalt.

A més dels definits per OMG, hi ha altres llenguatges que permeten definir DSL i dotar-los amb una sintaxi concreta textual. Un d'aquests és Xtext, un projecte d'Eclipse d'ampli ús pels mecanismes i eines que proporciona per a definir editors i validadors de manera automàtica.

Web recomanat

Per saber més sobre Xtext en podeu consultar la pàgina web: http://www.eclipse.org/Xtext

El principal avantatge de les notacions textuals és que permeten descriure expressions i relacions complexes entre els elements del model, amb més nivell de detall. No obstant això, els models descrits així de seguida es converteixen en immanejables i incomprensibles pels humans, a causa de la seva extensió i complexitat.

En la pràctica la millor solució és utilitzar una combinació de notacions gràfiques i textuals, amb la qual cosa és possible obtenir els avantatges de totes dues opcions. D'aquesta manera la representació d'alt nivell del sistema es fa utilitzant una notació gràfica, mentre que les propietats de gra fi, que cal descriure amb més detall, s'especifiquen utilitzant una notació textual.

L'acció d'assignar un significat a un símbol donat es coneix com a interpretació.

Nota

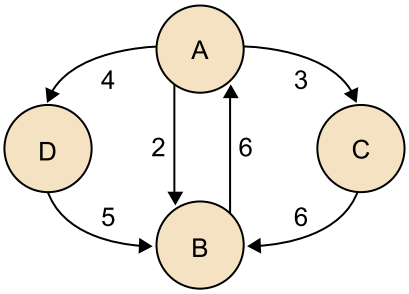
L'acció d'assignar un significat a un símbol determinat es coneix com a *interpretació*.

Hem comentat anteriorment que la sintaxi abstracta ha de ser independent de qualsevol notació i de qualsevol semàntica que se li assigni. Una cosa similar hauria d'ocórrer amb la sintaxi concreta, però en la pràctica això és més complicat: encara que en teoria una sintaxi concreta és independent de qualsevol sintaxi abstracta i de qualsevol semàntica (significat), els humans normalment solem associar significats concrets a símbols que veiem, sobretot en certs contextos.

Per exemple, un rectangle sol significar una classe UML o una entitat en un diagrama d'entitat-relació, mentre que les línies representen relacions. Per la seva banda, els rombes s'associen a decisions en els models de flux, tant en UML com en altres notacions que representen processos (Flowchart, BPMN, etc.). Un exemple en què una imatge representa clarament la classe associada a aquesta ocorre amb el model de la figura 9: a un element de tipus `Hammer` se li ha assignat la figura d'un martell com a sintaxi gràfica.

El problema sorgeix quan dues persones assignen dos significats diferents a un mateix símbol.

Figura 12. Què significa aquest model?



El model mostrat en la figura 12 admet diverses interpretacions, totes vàlides:

1) Distàncies entre ciutats, expressades en quilòmetres.

2) Transicions entre estats d'un diagrama d'estats, en què els nombres indiquen el temps en mil·lisegons que es triga a dur a terme cada transició.

3) Fluxos de migració mensuals entre aeroports, expressats en milers de persones.

4) Deutes entre entitats bancàries, expressats en milions d'euros.

Consulta recomanada

Aquest exemple es deu al professor Gonzalo Génova (UC3M): "Modeling and Metamodeling in Model Driven Development - Part 1: What is a model: syntax and semantics". http://www.ie.inf.uc3m.es/ggenova/Warsaw/Part1.pdf

Exercici proposat

Definiu quatre metamodels, cadascun representant la sintaxi abstracta de les quatre interpretacions diferents del model esmentades en el text i comparar-los entre si. Quines són les diferències entre ells?

3.2.3. Semàntica

Tant la sintaxi abstracta com la sintaxi concreta permeten descriure models d'un sistema. No obstant això, com exemplifica la figura 12, la informació que proporcionen aquestes dues sintaxis pot no ser suficient per a comprendre de manera precisa el significat d'un model, ni per a poder raonar sobre aquest i sobre el seu comportament. Per tant, la definició de qualsevol llenguatge ha de proporcionar també informació sobre la seva **semàntica**, a més de sobre les seves sintaxi abstracta i concreta. Sense aquesta informació pot ser que dues persones (o eines) interpretin i manipulin un mateix model de dues maneres diferents, la qual cosa pot portar a malentesos, raonaments incorrectes sobre el sistema o a implementacions errònies.

Les operacions admissibles sobre el model de la figura 12 (i també el seu comportament) depenen molt de la interpretació que en fem. Així, si el model representés el deute entre entitats bancàries, podríem reduir aquest mateix model a un altre d'equivalent, en el qual el banc A degui 3 milions a B, D en degui 1 a B, i el banc C en degui 3 a B, mitjançant successives redempcions dels deutes (si A deu 2 a B, i B deu 6 a A, això és equivalent al fet que B en degui 4 a A). No obstant això, això no tindria cap sentit si el model de la figura 13 representés transicions d'una màquina d'estats, o distàncies entre poblacions. Tampoc no tindria sentit en aquest cas que el graf admetés arcs d'un node a si mateix, cosa que sí que tindria molt sentit si el graf representés un diagrama d'estats.

En general, el que **sembla que representa** un model no és el mateix que el que **realment significa**.

Hi ha diverses maneres de definir la semàntica d'un llenguatge, cadascuna més apropiada al tipus d'ús que pretenguem donar a aquesta semàntica.

- La semàntica **denotacional** tradueix (o transforma) cada sentència o model del llenguatge en una sentència o model d'un altre llenguatge que té una semàntica ben definida. Aquesta manera de donar semàntica correspon a una interpretació o compilació del model, però en què el llenguatge destinació no sol ser un llenguatge de modelització o programació, sinó un formalisme matemàtic (denominat *domini semàntic*). Per a transformar un model d'un llenguatge en un model d'un altre llenguatge amb una semàntica ben definida s'utilitzen les transformacions horitzontals entre models, que es veuran en l'apartat 4.

Nota

Per dotar de semàntica denotacional un DSL se solen utilitzar transformacions de models que converteixen els models del llenguatge en models del formalisme destinació.

- La semàntica **operacional** descriu el comportament dels models del sistema de manera explícita, mitjançant un llenguatge d'accions o definint operacions i especificant-ne el comportament.

Consulta recomanada

S. Mellor; M. Balcer (2002). *Executable UML: A foundation for model-driven architecture*. Addison Wesley.

Webs recomanats

http://www.omg.org/spec/FUML/Current

http://www.omg.org/spec/ALF/Current

Implementació d'fUML: http://fuml.modeldriven.org

Parser per a Alf: http://lib.modeldriven.org/MDLibrary/trunk/Applications/Alf-Reference-Implementation/dist/

Presentacions: www.slideshare.net/seidewitz

En l'apartat 2 hem vist un exemple de semàntica operacional per als models de cadenes de muntatge, mitjançant el qual enriquíem el metamodel amb les operacions que admetien els diferents elements, i definíem el comportament d'aquestes operacions mitjançant precondicions i postcondicions expressades en OCL.

En UML també és possible especificar comportament mitjançant accions, i poder executar els models com si fossin programes. Això és possible només si la descripció del comportament està suficientment detallada, i a més hi ha una màquina o plataforma que sigui capaç d'executar tals especificacions de comportament. La primera proposta per a especificar un model UML executable la van fer Mellor i Balcer. Posteriorment Ed Seidewitz ha definit un subconjunt d'UML (denominat *fUML*) amb semàntica ben definida i un llenguatge d'accions per a ell (denominat *Alf, Action Language for fUML*), que han estat recentment estandarditzats per l'OMG. La sintaxi concreta d'Alf és textual, una barreja d'OCL amb les instruccions d'assignació i d'iteració comunes als llenguatges de programació imperatius.

- La semàntica **axiomàtica** descriu un conjunt de regles, axiomes o predicats que han de complir les sentències del llenguatge (és a dir, els models ben formats), i les interpreta en una lògica en què és possible raonar sobre aquestes. La lògica de Hoare per a llenguatges de programació és un exemple d'aquest tipus de semàntica, i també la lògica de reescriptura per a gramàtiques de grafs.

Per exemple, per a models com el de la figura 13, suposant que modelitzin deutes entre entitats bancàries, podem pensar en els quatre axiomes (o regles) següents que en defineixen el comportament i permeten derivar uns models a partir d'altres:

1) [Suma] Dos arcs que comparteixen el mateix origen i la mateixa destinació poden ser reemplaçats per un arc l'etiqueta del qual és la suma dels dos.

2) [Resta] Dos arcs amb orígens i destinacions oposats es poden reemplaçar per un arc orientat com el de valor més gran i etiquetat amb la diferència de les etiquetes.

3) [Nul] Un arc etiquetat amb 0 pot ser eliminat.

4) [Cicles] Les etiquetes d'un conjunt d'arcs que formin un cicle poden ser incrementades o disminuïdes en una mateixa quantitat (la regla 2 és un cas particular d'aquesta, tenint en compte també la regla 3; en altres paraules, la regla 2 pot ser derivada de les regles 3 i 4).

Aquestes regles defineixen el comportament del sistema mitjançant un sistema de reescriptura de grafs.

Hi ha nombroses eines que permeten especificar el comportament d'un sistema en termes de reescriptura o transformació de grafs, com per exemple GrGen.Net, ATOM3, FuJaBa, AGG, PROGRES, MotMot, e-Motions o Henshin. Cadascuna permet a més fer diferents tipus d'anàlisis (confluència, terminació, assolibilitat), simular o executar els models o fins i tot generar codi directament per a diferents plataformes com C++ o .NET.

A l'hora d'especificar la semàntica d'un llenguatge no és solament essencial que sigui precisa, sinó que a més sigui útil per a fer les tasques que es pretén que faci. Així, una descripció matemàtica del sistema serà molt útil per a raonar sobre el sistema si els dissenyadors la poden entendre, ja que permet fer les anàlisis que es pretenen, i tenim eines per a fer aquestes anàlisis. En un altre cas la seva utilitat no estaria justificada en absolut.

Per exemple, per a què volem una semàntica del nostre sistema definida en termes de xarxes de Petri estocàstiques si no som experts en aquest formalisme o no tenim eines per a manejar aquest tipus d'especificacions? Igualment, tenir una especificació operacional del sistema en terme de precondicions i postcondicions en OCL no és gaire útil si el que volem és analitzar algunes propietats com el rendiment (*throughput*) del sistema, la seva fiabilitat o altres característiques de qualitat de servei.

En general, no hi ha una manera millor o pitjor d'especificar la semàntica d'un llenguatge, sinó que depèn molt de l'ús que vulguem fer d'aquesta semàntica i del nostre coneixement sobre ella.

3.2.4. Relacions entre sintaxi abstracta, sintaxi concreta i semàntica

Per finalitzar aquesta secció destacarem algunes relacions entre la sintaxi abstracta, la sintaxi concreta i la semàntica d'un llenguatge, i també bones pràctiques per a un ús correcte.

- Mentre que la sintaxi abstracta especifica quins són els models vàlids, la sintaxi concreta permet representar aquests models. Per la seva banda, la semàntica els assigna un significat precís i no ambigu.

- En MBE, la manera natural de definir la sintaxi abstracta dels DSL és mitjançant metamodels.

- La relació entre la sintaxi concreta d'un llenguatge se sol definir a partir de la seva sintaxi abstracta, mitjançant una relació (*concrete syntax mapping*) que assigna un símbol de la sintaxi concreta a cada concepte de la sintaxi abstracta. Aquesta assignació ha de respectar la semàntica que solen tenir els símbols.

Exemple

Per exemple, el color vermell sol indicar perill o prohibició, no és apropiat per a representar permisos o obligacions. Igualment les línies solen representar arcs d'un graf o associacions entre elements, i no s'haurien d'utilitzar per a representar classes o entitats.

- Una mateixa sintaxi abstracta pot tenir associada més d'una sintaxi concreta, com per exemple una de textual i una altra de gràfica. En MDD, tots els DSL tenen almenys una sintaxi concreta, que és la que ofereix per defecte el llenguatge en el qual estan definits els seus metamodels.

- El *concrete syntax mapping* sol estar definit per dues funcions: de la sintaxi concreta a la sintaxi abstracta (el que es coneix com a *parsing function*) i de la sintaxi abstracta a la concreta (*rendering function*).

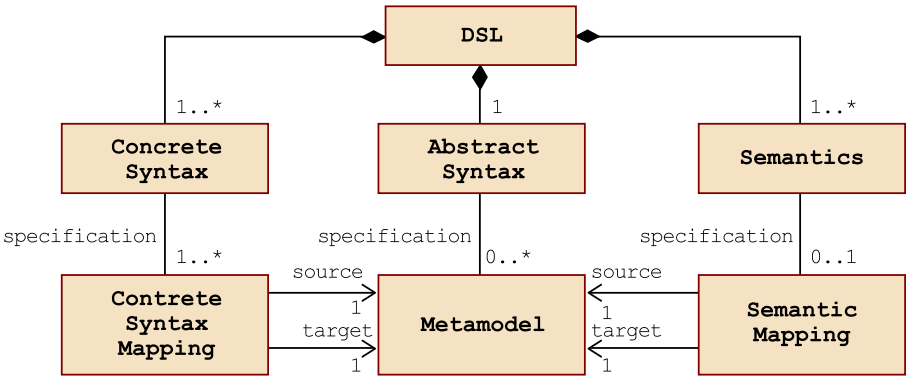
- Quant a la semàntica, a un llenguatge li podem assignar diversos significats, cadascun definit per la seva interpretació en un domini semàntic concret. Cada interpretació està descrita per una transformació (*mapping*) entre el metamodel de la seva sintaxi abstracta i el metamodel del domini semàntic destinació, que assigna un significat a cadascun dels models vàlids del DSL.

- El tipus de semàntica que cal definir per a un llenguatge dependrà de l'ús que se'n vulgui fer, i del tipus de raonaments i eines disponibles en el do-

mini semàntic destinació. Sempre és possible definir diverses interpretacions per a un mateix llenguatge, encara que en aquest cas cal cuidar la *consistència* entre aquestes (de manera que tot el que s'infereixi en un domini semàntic sigui cert també en la resta, i no hi hagi contradiccions).

En MDE, els *mappings* se solen definir i implementar mitjançant transformacions de models. La figura 13 mostra a manera de resum tots aquests conceptes d'una manera gràfica. Les seccions següents aprofundeixen una mica més en alguns aspectes concrets.

Figura 13. Parts d'un DSL i les relacions entre aquestes



3.3. Metamodelització

Hem vist la importància que tenen els metamodels per a representar els conceptes d'un llenguatge, les relacions entre aquests i les regles estructurals que restringeixen els possibles elements dels models vàlids. Per això els metamodels es converteixen en artefactes essencials d'MDE.

Ja hem comentat en l'apartat 1 que els metamodels són al seu torn models, i per tant estan definits pel seu metametamodel. Segons la proposta d'OMG, aquesta estructura recursiva (denominada *arquitectura de metamodelització*) acaba en el nivell 3, ja que els metametamodels es poden usar per a definir-se a si mateixos.

Vegeu també

Vegeu la figura 1 de l'apartat 1.

Aquesta arquitectura de metamodelització és una peça clau en MDA. De fet, l'existència d'un metametamodel com MOF és el que permetrà definir de manera unificada tots els metamodels dels diferents llenguatges de modelització que intervenen en MDA, i per tant fer-los compatibles entre si.

Per descomptat, a més de MOF hi ha altres propostes per a implementar les tècniques d'MDE, com per exemple EMF (*Eclipse modeling framework*) d'Eclipse, DSL Tools de Microsoft, GME (*generic modeling environment*) de la Universitat de Vanderbilt, o la plataforma AMMA de l'INRIA i la Universitat de Nantes. Cadascun defineix el seu metametamodel propi per a definir els seus llenguatges, com per exemple Ecore per a EMF o KM3 per a AMMA. L'avantatge que aquests artefactes de programari siguin al seu torn models en permet la interconnexió mitjançant transformacions de models, i s'aconsegueix així tendir ponts entre aquests anomenats *espais tècnics* (*technical spaces*).

Lectura recomanada

H. Brunelière; J. Cabot; C. Clasen; F. Jouault; J. Bézivin (2010). "Towards Model Driven Tool Interoperability: Bridging Eclipse and Microsoft Modeling Tools". *Proc. of ECMFA 2010*.

Quant a l'especificació del metamodel d'un sistema, és una tasca que no és simple i que en general ha de ser escomesa per equips multidisciplinaris que combinin experts del domini de l'aplicació que coneixen el vocabulari i notacions habituals amb experts en modelització conceptual que sàpiguen representar aquests conceptes en termes de metamodels concrets.

Consultes recomanades

Per a saber més sobre la definició de DSL es recomanen les lectures següents: Mernik i altres, 2005; Strembeck i Zdun, 2009; Kelly i Tolvanen, 2008; Cook i altres, 2007; Fowler, 2011, i Kleppe, 2008.

Finalment, sempre que es parla de metamodels sorgeix la seva relació amb les **ontologies**, que són un altre dels mecanismes disponibles actualment per a representar dominis d'aplicació de manera independent de les plataformes tecnològiques i de les implementacions subjacents. Des del nostre punt de vista, la principal diferència entre un metamodel i les ontologies és que aquestes últimes no solament permeten capturar els conceptes del domini i les relacions entre aquestes, sinó que també fan ús d'un sistema d'inferència (normalment basat en lògica de primer ordre) per a dotar de certa semàntica aquests elements i raonar sobre ells.

Per exemple, solen distingir entre diferents tipus d'associacions ("is-a", "is-part-of", "refines"...) amb significats ben definits.

Així, a més del que es pot expressar amb un metamodel, amb una ontologia també és possible obtenir noves relacions derivades de les existents, comprovar la consistència entre relacions específiques, etc.

Nota

Una ontologia és un conjunt de conceptes, les seves definicions i les relacions entre aquests, que permeten expressar idees vàlides en un domini determinat.

Per exemple, si un model A està relacionat amb un model B mitjançant una associació de tipus "is-a", i B està relacionat amb un tercer model C per aquest mateix tipus d'associació, llavors es pot concloure que A també està relacionat amb C per una associació "is-a", a causa de la propietat de transitivitat d'aquest tipus d'associació.

D'altra banda, les ontologies no han de tenir necessàriament llenguatges precisos i ben definits per a ser descrites, com ocorre amb els metamodels, que han d'estar descrits en el llenguatge del seu metametamodel, a més de ser capaços de definir models vàlids conformes a la sintaxi abstracta que defineixen.

Lectura recomanada

W. Pidcock (2003). "What are the differences between a vocabulary, a taxonomy, a thesaurus, an ontology, and a meta-model?" http://infogrid.org/wiki/Reference/PidcockArticle

En aquesta secció definirem primer la sintaxi concreta per a metamodels i en donarem algunes propostes. A continuació, descriurem una d'aquestes propostes, que és la que ofereix OMG per a definir la sintaxi concreta de DSL interns mitjançant extensions del metamodel d'UML, la qual cosa permet definir notacions gràfiques fent ús de les eines de modelització UML existents. Per descomptat, hi ha altres alternatives, i aquesta proposta té avantatges i inconvenients quan la comparem amb aquestes, les quals tractarem d'analitzar al llarg d'aquesta secció.

3.3.1. Sintaxi concreta per a metamodels

Una vegada tenim el metamodel que defineix la sintaxi abstracta d'un domini d'aplicació per a modelitzar sistemes d'aquest domini, cal definir una sintaxi concreta per a aquest. Aquesta sintaxi concreta proporcionarà la notació que permeti als dissenyadors descriure els models.

Tal com hem comentat en la secció 3.2.2, hi ha diferents tipus de sintaxi concreta (textual, gràfica) i diferents propostes i eines per a definir-les.

Així mateix, la sintaxi concreta es pot definir des de zero, o bé fent ús d'una notació existent, estenent-la amb els conceptes del nostre llenguatge. Això distingeix dos tipus de llenguatges específics de domini: interns i externs.

Un **DSL intern** utilitza la notació d'un llenguatge existent (el llenguatge "amfitrió") com a base per a definir els seus conceptes i la seva sintaxi concreta.

Per exemple, en la secció següent veurem com podem definir llenguatges específics de domini estenent UML amb els nous conceptes del nostre domini. En aquest sentit, UML es comporta com a llenguatge amfitrió. Això és molt útil i còmode perquè es pot reutilitzar tant la sintaxi general d'UML com les seves eines de modelització. També és possible definir DSL interns amb llenguatges textuals, i per exemple Ruby és un dels més utilitzats per a aquestes tasques per les facilitats d'extensió que proporciona. Lisp és un altre llenguatge que s'ha utilitzat per a definir DSL interns, i també els llenguatges de programació Java i C#. Els DSL interns es denominen també *DSL encastats* (*embedded*).

Exemple

Un exemple de DSL intern construït sobre Ruby és RubyTL, un llenguatge molt potent per a la transformació de models: http://rubytl.rubyforge.org/

Un **DSL extern** defineix la seva sintaxi concreta de manera independent de qualsevol altre llenguatge.

L'avantatge dels DSL externs és que no depenen d'altres, i permeten ser més petits, flexibles i centrats en el domini. D'altra banda, obliguen a haver de desenvolupar totes les eines per a definir-los i manipular-los, com per exemple editors, *parsers*, validadors, etc. No obstant això, cada vegada hi ha al mercat més eines que permeten fer aquestes tasques tant per a DSL externs gràfics com textuals, tal com esmentarem més endavant al final de la secció "Definició de plataformes".

3.3.2. Maneres d'estendre UML

El fet que UML sigui un llenguatge de propòsit general proporciona una gran flexibilitat i expressivitat a l'hora de modelitzar sistemes. No obstant això, hi ha nombroses vegades en les quals és millor tenir algun llenguatge de propòsit més específic per a modelitzar i representar els conceptes de certs dominis particulars. Això succeeix quan:

- la sintaxi o la semàntica d'UML no permeten expressar els conceptes específics del domini, o

- quan volem restringir i especialitzar els constructors propis d'UML, que solen ser massa genèrics i nombrosos.

OMG defineix tres possibilitats a l'hora de definir llenguatges específics de domini (vegeu la figura 14):

Vegeu també

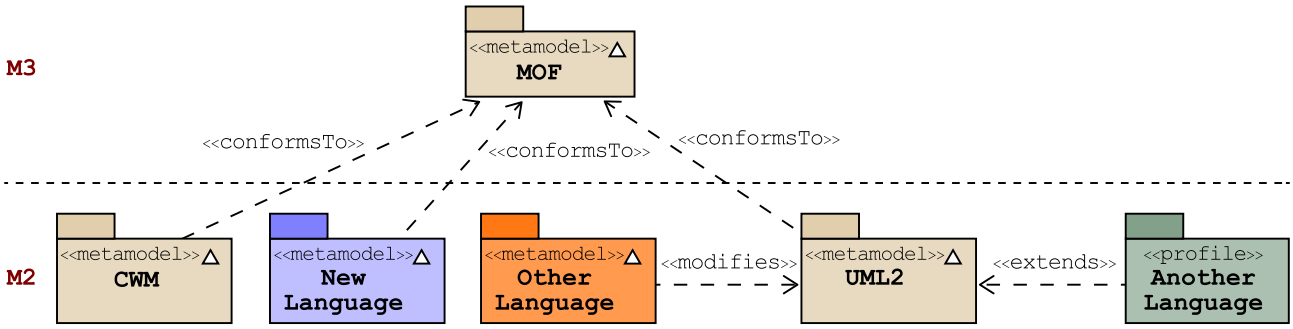
MOF és el llenguatge de metamodelització definit per l'OMG per a descriure llenguatges de modelització. Vegeu els apartats 1.2.7 i 3.3.

a) Es defineix un llenguatge completament nou a partir de MOF.

b) Es modifica el metamodel d'UML, afegint noves classes, modificant-ne altres o esborrant-ne algunes. Això es coneix com una extensió "pesant" d'UML (*heavyweight extension*).

c) S'estén l'UML mateix, especialitzant alguns dels seus conceptes i restringint-ne altres, però respectant la semàntica original dels elements d'UML (classes, associacions, atributs, operacions, transicions, etc.). És a dir, l'estructura original d'UML no es modifica, sinó que s'afegeixen coses (classes, relacions...) a partir d'aquesta. Això es coneix com una extensió "lleugera" d'UML (*lightweight extension*).

Figura 14. Maneres proposades per l'OMG per a definir llenguatges



En la figura 14, M2 i M3 corresponen als nivells homònims de la "torre de models de l'OMG", tal com es descriu en les figures 1 i 2 de l'apartat 1.

Web recomanat

Per a més informació sobre CWM (*common warehouse metamodel*) vegeu http://www.omg.org/cwm.

La primera opció és l'adoptada per llenguatges com CWM, ja que la semàntica d'alguns dels seus constructors no coincideix amb la d'UML i els seus creadors van preferir no basar-se en UML sinó definir un llenguatge nou. Per a això només cal descriure el metamodel del nou utilitzant MOF. En aquest cas estaríem parlant de DSL externs. El principal avantatge de definir un nou llenguatge d'aquesta manera és que permet més grau d'expressivitat i correspondència amb els conceptes del domini d'aplicació particular, igual que quan ens fem un vestit a mesura. No obstant això, el fet de no respectar el metamodel d'UML impedirà que es puguin utilitzar les eines UML existents en el mercat (editors, validadors, simuladors, etc.), ja que no poden manejar els nous conceptes.

La segona opció consisteix a tractar de reutilitzar el metamodel d'UML, adaptant-lo als nostres requisits particulars, però podent modificar les regles i restriccions d'integritat d'UML, i també la semàntica.

Per exemple, podríem voler poder definir associacions entre elements que no siguin classes, sinó també entre paquets o entre operacions. O que els elements dels nostres models puguin tenir diversos noms (sinònims) i no solament un.

El principal avantatge d'aquesta alternativa és que podem reutilitzar moltes parts d'UML tal com estan definides, la qual cosa representa no haver de redefinir alguns subconjunts del metamodel d'UML que són de molta utilitat (per exemple, màquines d'estats o diagrames de components). Ara bé, el resultat obtingut ja no és UML, i per tant en aquest cas també perdem la possibilitat d'usar les eines existents per a UML. També es perd la comprensió esperada dels diagrames UML, en haver canviat el seu significat.

Finalment, hi ha situacions en les quals és suficient amb estendre el llenguatge UML afegint elements o especialitzant els existents, però respectant-ne la semàntica. El llenguatge UML permet aquest mecanisme d'extensió utilitzant el que es denominen *perfils UML* (*UML profiles*).

El fet que aquestes extensions siguin conservatives (és a dir, que respectin la semàntica d'UML) ens permetrà usar els entorns o eines existents per a UML amb els nostres models, i els podrem crear, validar o intercanviar amb altres eines UML sense problemes.

En el cas de la segona i tercera opcions estaríem parlant de DSL interns, el llenguatge amfitrió dels quals és UML.

En general, resulta difícil decidir quan és millor crear un nou metamodel, modificar el d'UML o definir una extensió d'UML usant els mecanismes d'extensió estàndard definits amb aquest propòsit. Potser la possibilitat de poder usar les eines UML existents és un dels punts principals per considerar, perquè la generació d'editors i validadors per a models no és una tasca fàcil. No obstant això, l'aparició d'aquest tipus d'eines[6] han afavorit notablement la primera opció. Aquestes eines permeten definir metamodels i construir editors visuals d'una manera elegant i assequible. Fins i tot molts tenen altres eines MDA per a la generació de codi en diferents plataformes a partir d'alt nivell (com per exemple l'eina Acceleo, d'Obeo). D'altra banda, GMF (*grapical modelling framework*) és un projecte d'Eclipse per a la generació d'eines visuals per DSL. Encara que molt potent i expressiva, requereix un coneixement i grau d'experiència elevat i no resulta fàcil crear editors de DSL amb GMF.

[6]Com per exemple, DSL Tools (de Microsoft), MetaEdit+ (de Metacase) o Obeo Designer (d'Obeo).

Per les raons exposades i per ser la manera més utilitzada actualment, ens centrarem a continuació a analitzar els mecanismes d'extensió que proporciona UML per a ser estès mitjançant perfils UML. Així mateix, també discutirem la utilitat i rellevància d'aquests perfils UML en el context d'MDA, en què tenen un paper destacat.

3.3.3. Els perfils UML

Els perfils UML van aparèixer en la versió 1 d'UML, ja que ja des del principi es pensava en UML com una família extensible de llenguatges, més que com una notació única. En la versió 2 d'UML els perfils van ser modificats lleugerament per a millorar-ne alguns aspectes.

Per exemple, la manera d'accedir des de les metaclasses als estereotips i viceversa, o els mecanismes per a estendre i adaptar les metaclasses d'un metamodel qualsevol (i no solament el d'UML) a les necessitats concretes d'una plataforma, com pot ser .NET o J2EE, o d'un domini d'aplicació (temps real, modelització de processos de negoci, etc.).

Web recomanat

La relació completa dels perfils UML definits per l'OMG està disponible a http://www.omg.org/technology/documents/profile_catalog.htm

UML 2.0 esmenta diverses raons per les quals un dissenyador pot voler estendre i adaptar un metamodel existent, sigui el de l'UML mateix, el de CWM, o fins i tot el d'un altre perfil:

- Disposar d'una terminologia i vocabulari propis d'un domini d'aplicació o d'una plataforma d'implementació concreta[7].

(7)Per exemple, poder manejar dins del model del sistema terminologia pròpia d'EJB, com "Home interface", "entity bean", "JAR", etc

- Definir una sintaxi per a construccions que no disposen d'una notació pròpia (com succeeix amb les accions).

- Definir una nova notació per a símbols ja existents, més d'acord amb el domini de l'aplicació objectiu[8].

(8)Poder usar, per exemple, una figura amb un ordinador per a representar computadors en una xarxa en lloc del símbol per a representar un node que ofereix UML

- Afegir certa semàntica que no apareix determinada de manera precisa en el metamodel[9].

(9)Per exemple, la incorporació de prioritats en la recepció de senyals en una màquina d'estats d'UML, rellotges, temps continu, etc.

- Afegir restriccions a les existents en el metamodel, restringint-ne la manera d'utilització[10].

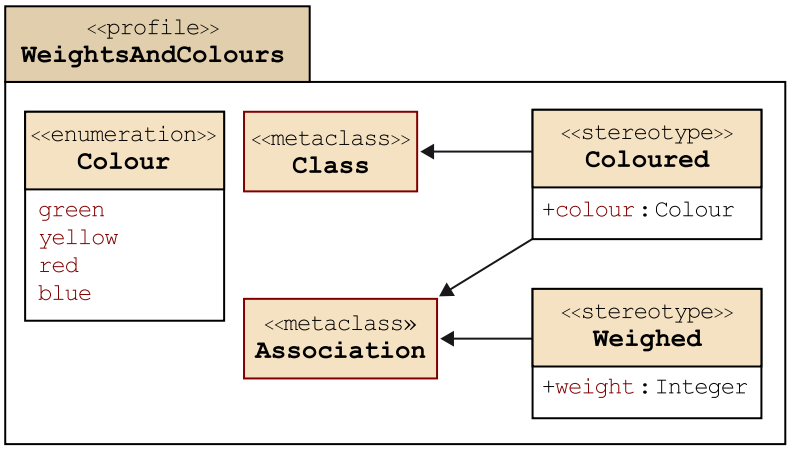
(10)Per exemple, impedint que certes accions s'executin en paral·lel dins d'una transició, o forçant l'existència de certes associacions entre les classes d'un model.

- Afegir informació que pot ser útil a l'hora de transformar el model a altres models o a codi.

Un perfil es defineix en un paquet UML, estereotipat "profile", que estén a un metamodel existent o a un altre perfil. S'utilitzen tres mecanismes per a definir perfils: estereotips (*stereotypes*), definicions d'etiquetes (*tag definitions*) i valors etiquetats (*tagged values*), i restriccions (*constraints*).

Per a il·lustrar aquests conceptes utilitzarem un petit exemple de perfil UML, que definirà dos elements nous que es poden afegir a qualsevol model UML: colors i pesos.

Figura 15. Un perfil UML per a estendre UML amb colors i pesos



En primer lloc, un estereotip està definit per un nom i per una sèrie d'elements del metamodel als quals es pot associar. Gràficament, els estereotips es defineixen com altres classes UML, però estereotipades (etiqueta `"stereotype"`). En el nostre exemple, el perfil UML `WeightsAndColours` defineix dos estereotips, `Coloured` i `Weighed`, que proporcionen color i pes a un element UML. Tots els estereotips estan associats a una o més metaclasses del metamodel d'UML, que són les que l'estereotip estén.

En l'exemple de la figura 15 (solament) les classes i les associacions d'UML es poden acolorir, i (solament) les associacions poden tenir associat un pes. Noteu com el perfil especifica els elements del metamodel d'UML (classes estereotipades "`metaclass`") sobre les quals es poden associar els estereotips que defineix el perfil mitjançant fletxes contínues de punta triangular negra.

Als estereotips és possible associar-los restriccions (*constraints*), que imposen condicions sobre els elements del metamodel que han estat estereotipats. D'aquesta manera es poden descriure, entre altres, les condicions que ha de verificar un model "ben format" d'un sistema en un domini d'aplicació. Per exemple, suposem que el metamodel del nostre domini d'aplicació imposa la restricció que si dues o més classes estan unides per una associació acolorida, el color de les classes ha de coincidir amb el de l'associació. Aquesta restricció es tradueix en la restricció següent del perfil UML:

```
context Coloured inv sameColour:
   self.base_Association.memberEnd->
      forAll(c | c.extension_Coloured->notEmpty()) implies
         c.extension_Coloured.colour=self.colour
```

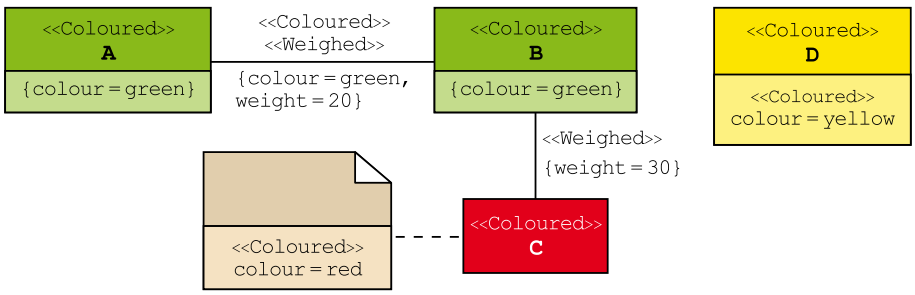
Noteu com és possible referir-se des d'un estereotip a la metaclasse que estén mitjançant `base_<M>`, en què `<M>` és el nom de la metaclasse (per exemple, `base_Class` o `base_Association`). Igualment, des d'una metaclasse és possible referir-se a un estereotip de nom `<X>` com a `extension_<X>`, i per tant preguntar si aquest estereotip està associat a la classe (la multiplicitat és [0-1] i, per tant, n'hi ha prou de preguntar si `notEmpty()`).

En l'exemple veiem com s'usa l'atribut `memberEnd` de la metaclasse `Association` d'UML per a referir-se a les classes relacionades mitjançant aquesta associació. Per a cadascuna es pregunta si està estereotipada com a `Coloured (c.extension_Coloured->notEmpty())` i, en aquest cas, es comprova si el color és el mateix que el de l'associació `(c.extension_Coloured.colour=self.colour)`.

Finalment, una definició d'etiqueta (*tag definition*) defineix un metaatribut addicional que s'associa a una metaclasse del metamodel estès per un perfil. Tota definició d'etiqueta ha de tenir un nom i un tipus, i es defineix dins d'un estereotip determinat. D'aquesta manera, l'estereotip `"Weighed"` del nostre exemple té un atribut denominat `"weight"`, de tipus `Integer`, que indica el pes de cada associació que hagi estat estereotipada com a `"Weighed"`. Les definicions d'etiquetes es representen de manera gràfica com a atributs de la classe que defineix l'estereotip. Quan un estereotip s'aplica a un element del model, els valors de les definicions d'etiquetes es defineixen mitjançant valors etiquetats (*tagged values*) que no són sinó parells (`etiqueta=valor`). Per exemple, `weight=30`.

La figura 16 mostra un exemple de l'ús d'aquest perfil sobre un model UML, en què algunes classes han estat estereotipades com a acolorides, i a les associacions se'ls ha afegit informació sobre el seu pes i color.

Figura 16. Un exemple d'ús del perfil UML amb pesos i colors



La figura mostra com és possible afegir més d'un estereotip a un element, i també mostrar els *tagged values* de diferents maneres: com a restriccions (el cas de la classe marcada amb color verd), en una subcaixa (com es mostra en la classe marcada com a groga) o en una nota a part (com en el cas de la classe marcada amb el color vermell).

És important assenyalar que aquests tres mecanismes d'extensió no permeten canviar o eliminar elements del metamodel que estenen, només afegir-los elements i restriccions, però respectant la seva sintaxi i semàntica original. Això és el que permet que les eines UML existents es puguin usar sense problemes amb els metamodels estesos per un perfil.

Actualment hi ha definits diversos perfils UML, alguns dels quals han estat fins i tot estandarditzats per l'OMG: els perfils UML per a CORBA i per a CCM (*CORBA component model*), el perfil UML per a EAI (*enterprise application integration*), el perfil UML per a MARTE (per a temps real i sistemes encastats),

el d'enginyeria de serveis (SysML) o el d'UPDM per a l'especificació de marcs arquitectònics d'aplicacions de defensa. Molts altres perfils UML es troben actualment en procés de definició i normalització per part de l'OMG.

Exemples

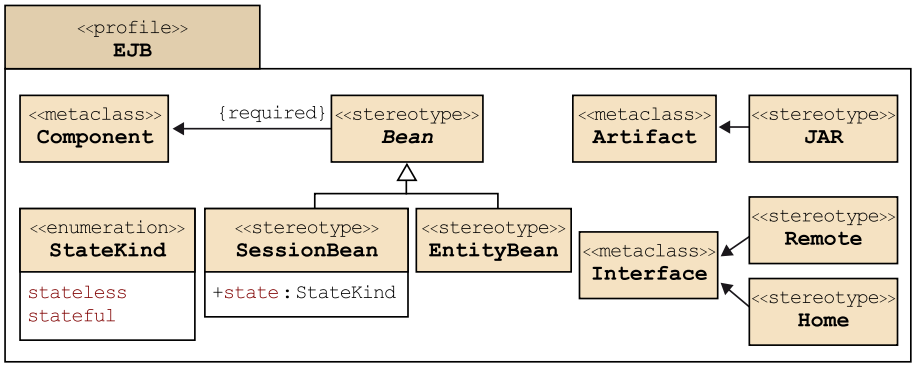
A més dels perfils UML que ha definit i estanderditzat OMG (vegeu http://www.omg.org/technology/documents/profile_catalog.htm) també hi ha perfils UML definits per altres organitzacions i empreses que, encara que no són estàndards oficials, estan disponibles de manera pública i són utilitzats normalment (es converteixen, per tant, en estàndards *de facto*). Un exemple de tals perfils és el definit per ISO i ITU-T per a l'especificació de sistemes d'informació que segueixen l'estàndard RM-ODP. També es disposa de perfils UML per a alguns llenguatges de programació, com poden ser Java o C#. Cadascun d'aquests perfils defineix una manera concreta d'usar UML en un entorn particular. Així, per exemple, el perfil UML per a CORBA defineix una manera d'usar UML per a modelitzar interfícies i artefactes de CORBA, mentre que el perfil UML per Java defineix una manera concreta de modelitzar codi Java usant UML.

Vegeu també

Consulteu l'estàndard "Reference Model of Open Distributed Processing (RM-ODP)" en l'assignatura *Enginyeria del programari de components i sistemes distribuïts* del grau d'Enginyeria Informàtica.

A manera d'exemple, la figura 17 mostra un perfil simple per a EJB, amb alguns dels conceptes d'aquest model de components (*session bean, entity bean*, *JAR*, *remote interface*, *homeinterface*) i les metaclasses d'UML que estenen aquests estereotips.

Figura 17. Un perfil UML molt simple per a EJB



3.3.4. Com es defineixen perfils UML

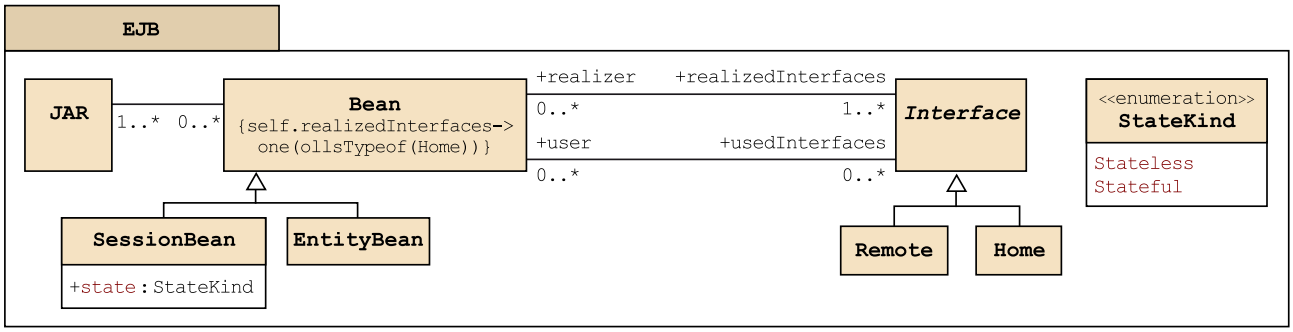
En aquesta secció descriurem un mètode d'elaboració possible de perfils UML. L'especificació d'UML 2.0 només es limita a definir el concepte de perfil i els seus continguts, sense descriure com s'elaboren. No obstant això, també és necessari poder disposar de certes guies i bones pràctiques que serveixin d'ajuda a l'hora de definir un perfil UML per a una plataforma o domini d'aplicació concret. Per això, a l'hora de definir un perfil UML es poden seguir els passos següents:

1) Abans de començar, cal disposar de la definició corresponent del metamodel de la plataforma o domini d'aplicació per modelitzar amb un perfil. Si no existís, llavors definiríem aquest metamodel utilitzant els mecanismes de MOF o de l'UML mateix (classes, relacions d'herència, associacions, etc.) de la ma-

nera usual. Haurem d'incloure la definició de les entitats pròpies del domini, les relacions entre aquestes, i també les restriccions que limiten l'ús d'aquestes entitats i de les seves relacions.

Per exemple, un possible metamodel per a modelitzar sistemes de components amb EJB és el mostrat en la figura 18, que inclou una restricció OCL que expressa que un `Bean` ha d'implementar exactament una interfície `Home`. Aquest metamodel és el que donarà lloc al perfil mostrat en la figura 17.

Figura 18. Un metamodel per a sistemes EJB



2) Una vegada disposem del metamodel del domini, passarem a definir el perfil. Dins del paquet "`profile`" inclourem un estereotip per a cadascun dels elements del metamodel que volem incloure en el perfil. Aquests estereotips tindran el mateix nom que els elements del metamodel, i s'establirà d'aquesta manera una relació entre el metamodel i el perfil. En principi qualsevol element que haguéssim necessitat per a definir el metamodel pot ser etiquetat posteriorment amb un estereotip.

3) És important tenir clar quins són els elements del metamodel d'UML sobre els quals aplicarem els diferents estereotips[11]. Cada estereotip s'aplicarà a la metaclasse d'UML que capturi millor la semàntica de l'element corresponent del metamodel del domini.

[11]Un exemple d'aquests elements són les classes, associacions, atributs, operacions, interfícies, transicions, paquets, etc

4) Proporcionar definicions d'etiquetes dels elements del perfil corresponents als atributs que apareguin en el metamodel. Incloure la definició dels seus tipus i els possibles valors inicials.

Això és el que succeeix amb l'atribut `state` de la classe `SessionBean` del metamodel, que es converteix en una definició d'etiqueta del mateix nom de la classe `SessionBean` del perfil.

5) Definir les restriccions que formen part del perfil, a partir de les restriccions del domini. Per exemple, les multiplicitats de les associacions que apareixen en el metamodel del domini o les regles mateixes de negoci de l'aplicació s'han de traduir en la definició de les restriccions corresponents.

Per a establir aquestes restriccions sol caldre navegar pel metamodel d'UML, i per això cal conèixer-lo bé. Per exemple, la restricció esmentada anteriorment que un `Bean` ha d'implementar exactament una interfície `Home` es tradueix en el perfil UML en la restricció OCL següent sobre els components estereotipats com `Bean`:

Nota

En el cas del perfil per EJB que es va mostrar en la figura 18, l'estereoptip `Bean` estén la metaclasse `Component`, `JAR` la metaclasse `Artifact` i `Interface` es reutilitza d'UML. També es reutilitzen les relacions que defineix UML mateix entre `Component`, `Artifact` i `Interface`.

```
context Bean inv oneRealizedHomeInterface:
   self.base_Component.realizedInterfaces()->
      one(extension_Home->notEmpty())
```

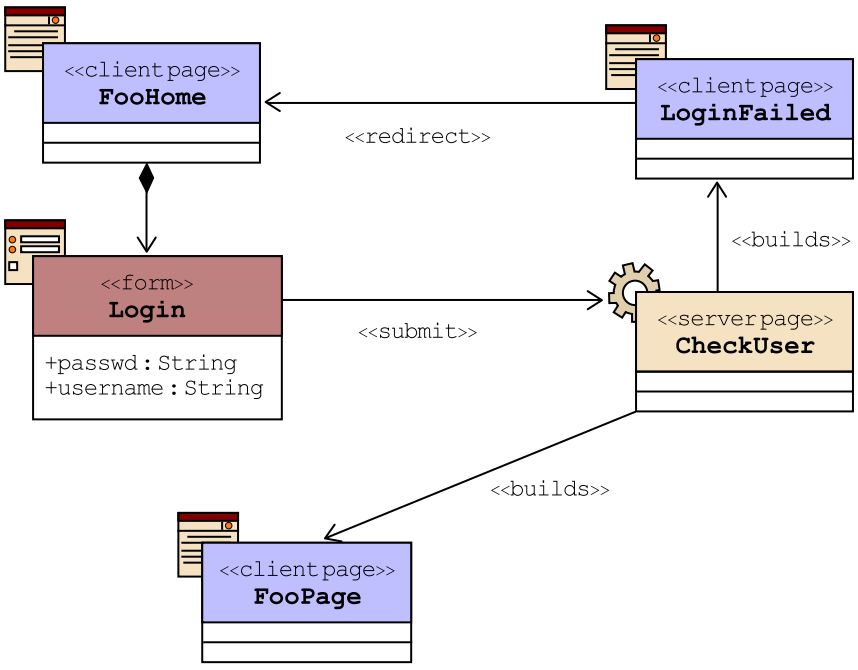
En aquesta expressió es fa ús d'una operació OCL definida en el metamodel d'UML (UML 2.3, §8.3.1, pàg. 152) que calcula el conjunt d'interfícies implementades per un classificador:

```
context Classifier def: realizedInterfaces () : Set(Interface) =
   self.clientDependency->
      select(dependency | dependency.oclIsKindOf(Realization) and
            dependency.supplier.oclIsKindOf(Interface)))->
               collect(dependency|dependency.client)
```

Finalment, cal esmentar que és possible associar una icona a cada estereotip, el qual serà mostrat en lloc de la icona estàndard que defineix UML per a la metaclasse estesa. Gràcies a això serà possible, per exemple, usar icones amb formes de màquines, martells i safates per a representar la figura 9 amb la cadena de muntatge.

Un altre exemple interessant de perfil UML és el definit per Jim Conallen per a modelitzar aplicacions web, i que proporciona conceptes com "*client page*", "*server page*", "*target*", "*javascript*" o "*form*" com a elements natius del llenguatge i permet usar-los des de qualsevol eina UML. La figura 19 mostra un model UML que fa ús d'aquest perfil per a modelitzar una part del sistema que demana entrada amb autenticació abans de permetre accedir a la pàgina "FooPage".

Figura 19. Exemple d'ús del perfil UML per a modelitzar aplicacions web



Consulta recomanada

J. Conallen (2003). *Building Web Applications with UML* (2a. ed.). Addison Wesley.

3.4. MDA i els perfils UML

A més de servir per a definir nous DSL a partir del metamodel d'UML, els perfils UML tenen un paper molt important en MDA per a fer dues tasques principals.

3.4.1. Definició de plataformes

En primer lloc, els perfils UML proporcionen mecanismes molt adequats per a descriure els models de les plataformes d'implementació, tal com hem vist de manera molt succinta amb el perfil per a EJB. A partir d'aquests models és possible establir una correspondència entre cadascun dels elements del model PIM i els de la plataforma, la qual determina de manera unívoca la transformació del PIM al PSM corresponent.

La idea és utilitzar els estereotips del perfil d'una plataforma per a "marcar" els elements d'un model PIM d'un sistema i produir el model PSM corresponent, ja expressat en termes dels elements de la plataforma destinació. Una marca representa un concepte en el model PSM, que s'aplica a un element del model PIM per a indicar com s'ha de transformar aquest element en el model PSM destinació. Les marques poden especificar també requisits extrafuncionals o de qualitat de servei sobre la implementació final[12].

(12)Per exemple, alguns dels elements del model PIM es poden marcar com a persistents, o subjectes a determinades condicions de seguretat.

El conjunt de les marques i les regles de transformació que en governen l'ús han de ser especificades de manera estructurada i normalment es proporcionen juntament amb el perfil UML de la plataforma destinació. Si el perfil UML de la plataforma inclou l'especificació d'operacions, llavors les regles de transformació haurien d'incloure també el tractament d'aquestes operacions.

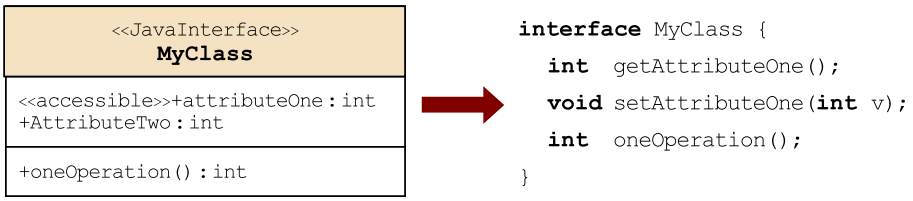
3.4.2. Marcatge de models

El segon ús dels perfils és per a "decorar" (o "marcar" segons la terminologia MDA) un model amb informació addicional sobre certs elements, per a indicar decisions de disseny o implementació sobre aquests.

> En MDA, una **marca** representa un concepte del PSM que pot ser aplicat a un element del PIM per a indicar com cal transformar aquest element.

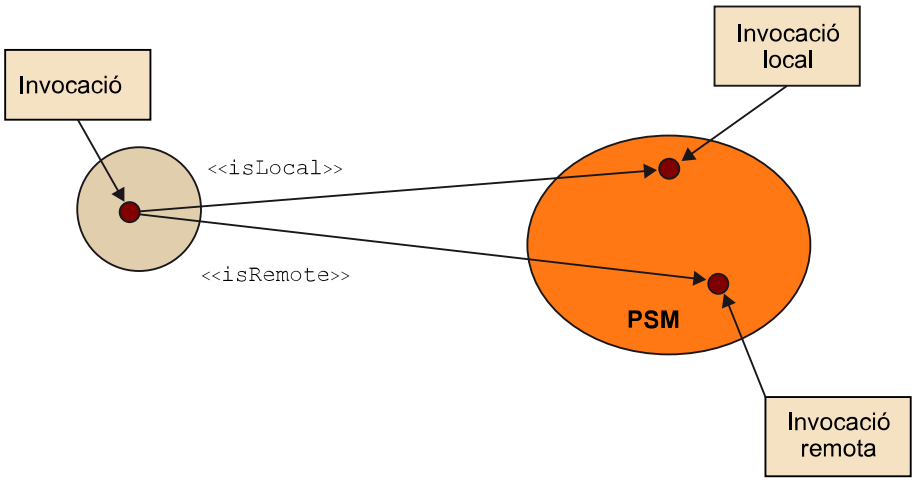
Per a il·lustrar aquest concepte, la figura 20 mostra una classe del PIM denominada `MyClass` amb dos atributs i una operació, que ha estat marcada amb dos estereotips, "`Java Interface`" i "`accessible`", per a transformar-la adequadament en un model PSM. El primer estereotip serveix per a indicar les classes del PIM que s'han de transformar en interfícies Java, i el segon permet seleccionar els atributs d'aquestes classes que han de ser visibles en el PSM. Com es pot observar en la figura, el segon atribut no apareix en el PSM transformat.

Figura 20. Un exemple de PIM marcat i la seva transformació a Java



De manera anàloga, la figura 21 mostra gràficament l'ús d'estereotips per a indicar en el PIM si una operació s'ha de transformar en una altra, la invocació de la qual s'ha de fer de manera local o remota en el PSM.

Figura 21. Un altre exemple de marques en el PIM per a decidir com s'han de transformar elements concrets en el PSM corresponent



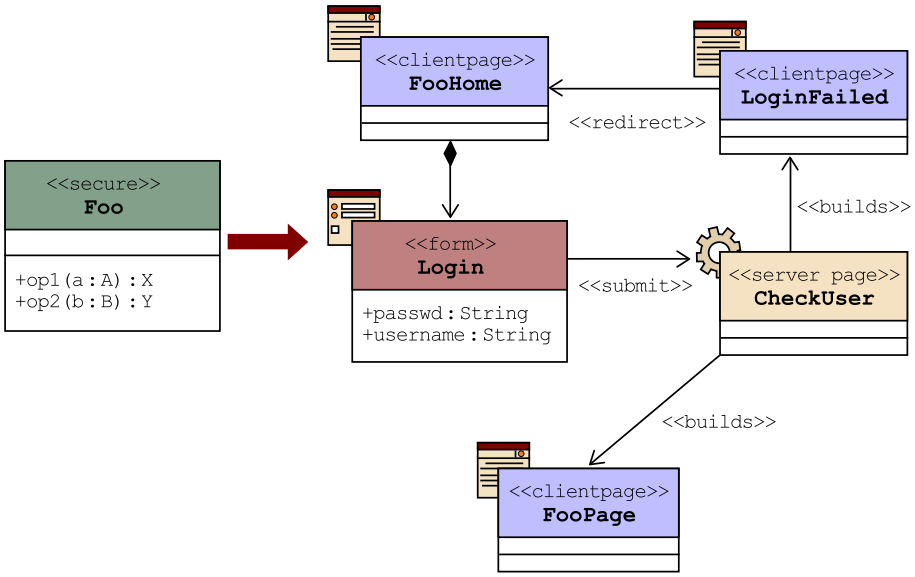
3.4.3. Plantilles

A més de marques, MDA també defineix el concepte de plantilla (*template*):

> En MDA, una **plantilla** (*template*) és un model parametritzat que especifica la forma i contingut de certes transformacions.

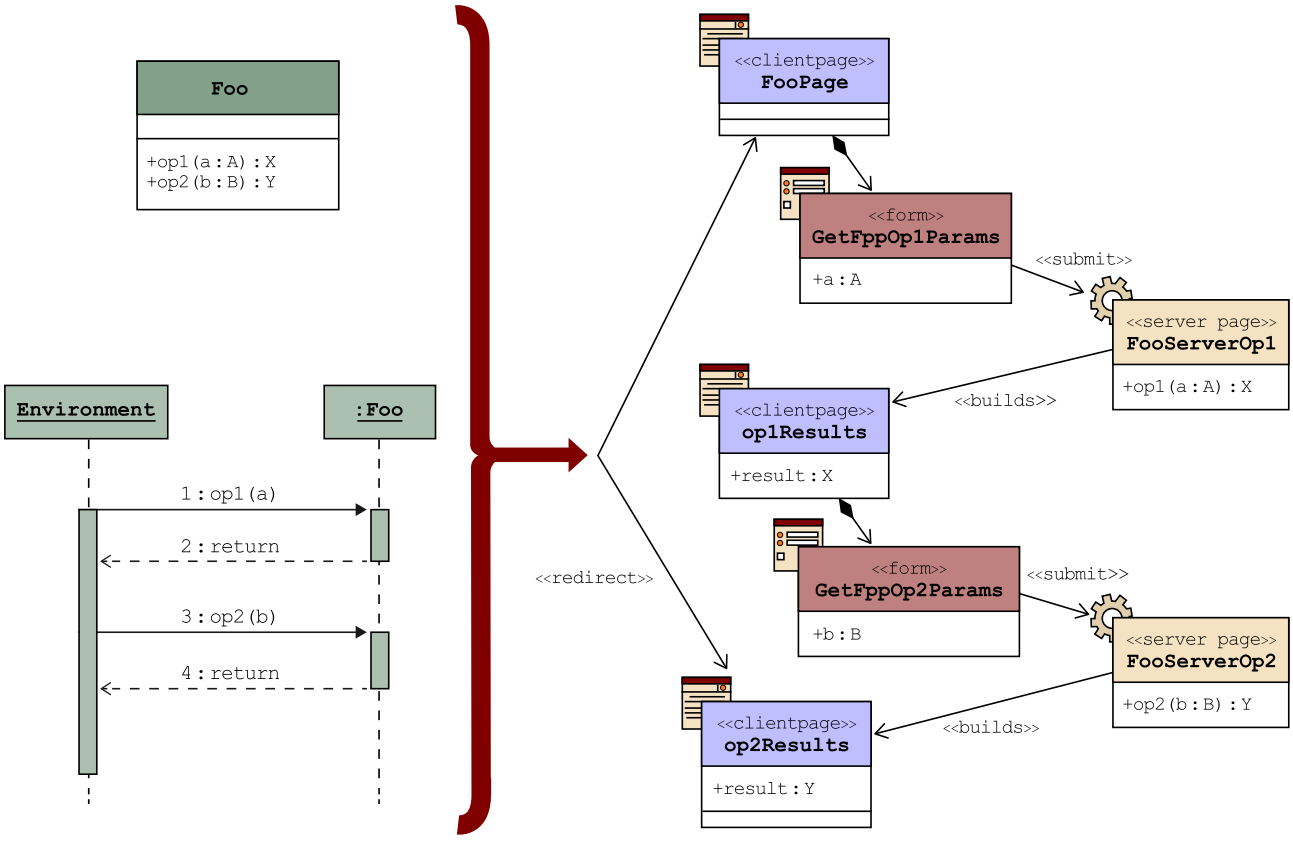
Per exemple, la figura 22 mostra el comportament d'una transformació de models que permet transformar classes marcades com a "`secure`" (`Foo`, en l'exemple) en un conjunt de classes que representen la pàgina web corresponent (`FooPage`) al costat de les pàgines necessàries per a autenticar els usuaris abans d'accedir-hi.

Figura 22. Un exemple de plantilles per a transformar models



La figura 23 mostra un altre exemple de l'ús de plantilles per a definir la transformació de determinats elements dels models marcats amb estereotips en conjunts d'elements del model destinació. En aquest cas, el PIM proporciona informació sobre una classe i les seves operacions, i també la seqüència vàlida en la qual s'han d'invocar aquestes operacions. A partir d'aquesta informació una transformació de models és l'encarregada de generar el model de l'aplicació web que invoca aquestes operacions des de la pàgina inicial corresponent.

Figura 23. Un altre ús de plantilles per a transformar models en MDA



3.5. Consideracions sobre la definició de DSL

Tal com hem esmentat anteriorment, hi ha nombroses alternatives per a definir DSL, o bé usar llenguatges existents per a expressar els models d'un sistema. En aquesta secció analitzem les principals opcions, i discutim els avantatges i inconvenients de cadascuna.

3.5.1. Ús de DSL enfront de llenguatges d'ús general ja coneguts

La primera decisió per considerar és si val la pena definir un nou DSL o usar algun llenguatge de modelització o programació existent.

Entre els avantatges de definir un nou DSL destaquem les següents:

- Els DSL permeten expressar el disseny del sistema usant una notació molt propera i natural per als experts del domini, amb conceptes que són els que manegen normalment. Això els permet entendre, escriure i validar els seus models propis.

- Els DSL permeten incrementar la usabilitat, mantenibilitat i reutilització dels dissenys i models per part dels experts del domini, de nou perquè usen una notació coneguda per ells.

- Els DSL permeten expressar sistemes de manera independent de la tecnologia usada o dels llenguatges d'implementació, i al nivell d'abstracció adequat.

- La validació dels models es pot fer des del domini del problema mateix, ja que un DSL pot controlar que es construeixin models vàlids.

- Els DSL solen ser molt més "compactes" i menys complexos que els llenguatges de modelització de propòsit general, amb moltes menys classes i relacions entre elles (i que, en la majoria dels casos, no se solen utilitzar).

- El nivell d'expressivitat i precisió que s'aconsegueix amb un DSL sol ser molt més elevat que el que s'aconsegueix amb un llenguatge de propòsit general.

- En molts casos és més senzill desenvolupar eines (o integrar algunes de les existents) per a manejar o analitzar models desenvolupats amb DSL que amb llenguatges de propòsit general, precisament per la seva genericitat i ampli espectre.

Nota

De fet, la "regla del 80-20" diu que per al 80% dels dissenys UML no cal més del 20% del llenguatge. Penseu que el metamodel d'UML 2 té més de 800 metaclasses.

Alguns desavantatges de definir DSL de manera *ad hoc*:

- El cost d'haver d'aprendre un llenguatge diferent per a cada domini d'aplicació no és menyspreable, especialment si el domini d'aplicació és bastant restringit.

- El disseny, implementació i manteniment d'un DSL no són tasques fàcils. I més si també hem de dissenyar, implementar i mantenir moltes de les eines associades a cada nou llenguatge (editors, validadors, etc.).

- Els mecanismes i eines existents per a reutilitzar (subconjunts de) llenguatges i combinar-los entre si són encara molt rudimentaris, la qual cosa obliga a començar gairebé des de zero cada vegada que volem definir i dissenyar un nou llenguatge.

- Encara que es guanyi notablement en expressivitat, pot ser que es perdi en altres aspectes, com eficiència o reutilització, per exemple.

- La definició descontrolada de nous DSL pot produir incompatibilitats entre empreses que defineixin DSL diferents per a modelitzar el mateix tipus de sistemes, o fins i tot que una mateixa empresa defineixi dues DSL diferents (i incompatibles entre si) per a modelitzar el mateix.

- L'experiència mostra que no tots els experts d'un domini usen de la mateixa manera els conceptes del domini, i fins i tot els poden usar de manera diferent. Això implica possibles problemes d'interoperabilitat i consistència entre models (i DSL) desenvolupats per diferents experts.

- No tots els experts d'un domini estan disposats (o preparats) a escriure i modificar models usant aquests DSL.

- Hi pot haver més dificultat per a integrar diferents DSL amb els sistemes i aplicacions d'una companyia que si s'usen llenguatges de propòsit general.

- La documentació i exemples existents sobre el disseny, desenvolupament i ús de DSL és bastant més escassa que l'existent sobre llenguatges de propòsit general.

3.5.2. Ús de perfils UML

Una alternativa interessant és definir un DSL mitjançant perfils UML, la qual cosa tracta de conjugar els avantatges de l'ús de DSL amb els que proporcionen els llenguatges de propòsit general com UML. No obstant això, l'ús d'una extensió lleugera com els perfils UML enfront de l'ús de definició de llenguatges a partir de MOF (o Ecore, en el cas d'Eclipse) tampoc no és una decisió fàcil de prendre, ja que totes dues opcions presenten avantatges i inconvenients.

Els perfils UML tenen alguns avantatges molt atractius:

- Són fàcils de definir i manejar si s'està acostumat a manejar models i eines UML.
- Hi ha un gran nombre de persones formades ja en UML. En aquest cas, es pot reutilitzar el coneixement i experiència existent en l'empresa sobre UML i sobre les seves eines.
- No cal oblidar que UML és un estàndard i això facilita enormement la interoperabilitat amb els sistemes d'informació i eines d'altres empreses.
- No cal aprendre altres llenguatges de modelització, ni adquirir altres eines.
- Són molt útils a l'hora de construir prototips de llenguatges d'una manera fàcil, ràpida i sense massa esforç.

D'altra banda, els perfils UML també presenten alguns inconvenients:

- La seva expressivitat és limitada, en haver de respectar la semàntica d'UML.
- En general són difícils de definir d'una manera rigorosa, sobretot per la complexitat que representa especificar les seves restriccions d'integritat sobre la base dels elements del metamodel d'UML.
- Els perfils no permeten "ocultar" res d'UML, per la qual cosa sempre s'arrossega tot el seu metamodel. Això té dos problemes principals: d'una banda, la complexitat del metamodel resultant, i de l'altra, que no es pot evitar que els usuaris del perfil usin la resta dels elements d'UML en els seus models, una cosa que sovint no volem.
- Manipular models desenvolupats amb perfils UML no és simple (per exemple, a l'hora de transformar-los), per la manera com es representen internament les extensions en UML.
- Els models desenvolupats amb perfils UML hereten també alguns dels problemes d'UML mateix, com la seva falta de precisió.
- Finalment, la notació que s'aconsegueix finalment amb un perfil UML no és tan compacta i atractiva com la que es pot aconseguir amb un DSL, amb una sintaxi concreta desenvolupada a mesura[13].

[13] Encara que és veritat que el cost de definir la sintaxi concreta per a un perfil UML és significativament més petit que el cost que implica definir la sintaxi concreta d'altres DSL.

La taula següent mostra alguns consells que poden ajudar a prendre una decisió en un o l'altre sentit, depenent de les circumstàncies particulars del dissenyador:

És convenient definir el DSL a partir de MOF (o Ecore) quan...	És convenient usar perfils UML quan...
El domini d'aplicació està ben definit i consolidat, i els conceptes estan àmpliament acceptats.	El domini no és gaire estàndard o estable, i per tant cal fer prototips ràpids del llenguatge i de les eines associades.
No es necessita poder combinar fàcilment models d'aplicacions de diferents dominis.	Cal combinar fàcilment models de diferents dominis d'aplicació.
La semàntica del domini no és compatible amb la d'UML.	És possible respectar la semàntica (i *look-and-feel*) d'UML per a modelitzar sistemes d'aquest domini.
L'empresa no té massa experiència en UML ni amb les eines associades.	L'empresa té eines UML que normalment utilitza en els seus projectes, i experiència amb UML.
La sintaxi concreta que es necessita per a expressar els models és molt diferent a la d'UML o no es pot aconseguir amb els perfils UML.	La notació que cal per al llenguatge no és gaire diferent de la d'UML, i es pot obtenir a partir d'aquesta mitjançant petits canvis.
Les eines que es necessiten per a editar, manipular i analitzar els models no admeten una integració fàcil amb UML/XMI.	Les eines associades als models s'integren bé amb les eines disponibles en l'empresa per a manejar models UML.

3.5.3. Llenguatges de modelització enfront de llenguatges de programació

Una distinció que tradicionalment s'ha fet és la dels llenguatges de modelització enfront dels de programació. Normalment s'ha considerat que els llenguatges de modelització s'usen de manera informal, no tenen una semàntica precisa i no són executables. Per contra, els llenguatges de programació sempre s'han vist amb una semàntica molt precisa (la que proporciona el compilador) i executables, encara que de molt baix nivell.

No obstant això, amb l'arribada d'MDE aquesta distinció s'ha anat perdent i els dos tipus de llenguatges es poden veure com a similars: tots dos tenen els mateixos components (sintaxi abstracta, sintaxi concreta i semàntica), i l'executabilitat és una cosa que ja no és exclusiva dels llenguatges de programació, tal com hem esmentat anteriorment.

Exemple

Per exemple, la pàgina http://wiki.eclipse.org/MoDisco/Components/Java/Documentation descriu detalladament el metamodel del llenguatge Java, és a dir, la seva sintaxi abstracta.

Potser la diferència principal es troba actualment en el nivell de detall que permeten expressar aquests llenguatges: en general són de més alt nivell els llenguatges de modelització que els de programació. De fet, els llenguatges de modelització se solen utilitzar sobretot en tasques de disseny, mentre que els llenguatges de programació se solen usar per a tasques d'implementació. Aquesta és probablement la principal raó per la qual els llenguatges de modelització solen tenir notacions gràfiques o visuals, enfront de les notacions textuals dels llenguatges de programació. No cal oblidar la facilitat amb la qual es representen en els llenguatges visuals les relacions entre els elements d'un model, una cosa que resulta més complicada en els llenguatges de programació. Per contra, i com també hem esmentat anteriorment, descriure processos i instruccions de processament seqüencials sol ser més senzill i apropiat

usant llenguatges de programació. De totes maneres, la situació va canviant i ja veiem com els llenguatges de modelització com UML es van complementant amb notacions executables, alhora que llenguatges de programació com Java comencen a tenir assercions i elements de més nivell d'abstracció per a descriure sistemes.

A més del nivell d'abstracció, també és important destacar una limitació actual dels llenguatges de programació: la **unificació**. Aquesta és la capacitat d'aconseguir que diversos llenguatges de programació coexisteixin i es combinin de manera senzilla i coherent per a formar l'especificació integrada d'un mateix sistema. En aquest sentit, l'**arquitectura de metamodelització** que hem presentat en aquest capítol permet un mecanisme per a ajudar a solucionar aquest problema, en compartir els llenguatges de modelització un metamodel comú. La metamodelització, juntament amb l'ús de transformacions de models per a convertir uns models en altres (tant a escala horitzontal com vertical), són les claus en les quals es basa MDE, i en particular MDA, per a la modelització, anàlisi i desenvolupament de sistemes d'informació d'una manera sistemàtica, predictible, mesurable i rigorosa. Les transformacions de models les estudiarem en l'apartat següent d'aquest mòdul.

4. Transformacions de models

4.1. La necessitat de les transformacions de models

Com es va descriure en l'apartat 1, els models són peces clau en l'àmbit d'MDE (*model-driven engineering*). Per això, és necessari tenir mecanismes de manipulació de models. Les transformacions de models proporcionen els mecanismes que permeten especificar la manera de produir models de sortida a partir de models d'entrada.

El concepte de transformació en el món de la informàtica no és nou. De fet, les transformacions són una cosa fonamental en l'enginyeria del programari, i la computació es pot veure com un procés de transformació de dades.

Els processos de transformació de dades es refereixen a la transformació d'informació en general, i cobreixen des de canvis en la representació de les dades a canvis en les dades mateixes. Per exemple, un procés de transformació de dades pot agafar un fitxer amb una codificació de caràcters concreta i convertir-lo en un altre fitxer amb una altra codificació de caràcters diferent, normalment per a aconseguir la interoperabilitat d'aplicacions o sistemes diferents. Un altre procés pot agafar les dades d'una base de dades de clients i obtenir una altra base de dades amb la informació de les vendes de la companyia, a partir de les dades de la primera.

Consulta recomanada

Els sistemes de transformacions de dades (o programes) estan fora de l'àmbit d'aquest capítol. No obstant això, una bona referència és: **H. Parstsch; R. Steinbrüggen** (1983). "Program Transformation Systems". *ACM Comput. Surv.* (vol. 15, núm. 3, pàg. 199-236).

Entre les aplicacions de les transformacions de model en l'àmbit d'MDE ens trobem:

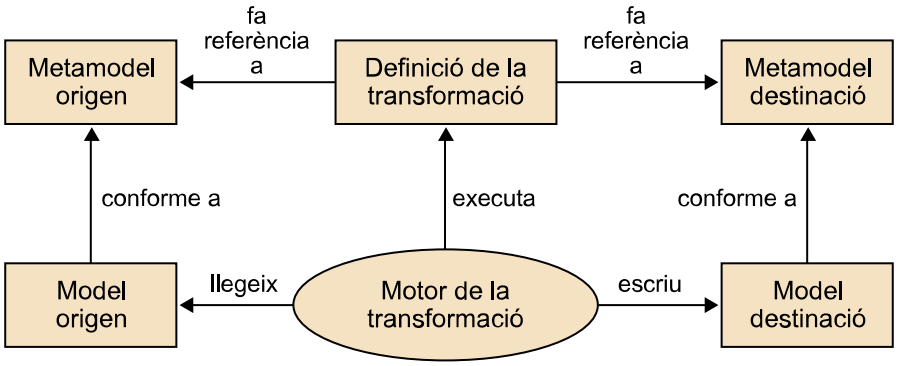
- Generació de models a més baix nivell, i finalment codi, a partir de models a alt nivell.
- Sincronització i mapatge entre models al mateix o diferents nivells d'abstracció.
- Creació de vistes basades en consultes sobre un sistema.
- Aplicació a l'enginyeria inversa per a obtenir models a més alt nivell a partir de models a més baix nivell.

4.2. Conceptes bàsics

Les **transformacions de models** es poden veure com a programes que prenen un model (o més d'un) com a entrada i retornen un altre model (o més d'un) com a sortida.

En la figura 24 es poden observar els participants involucrats en una transformació de models. S'hi mostra un escenari de transformació amb un model d'entrada i un de sortida, els quals han de ser conformes als seus metamodels respectius. Com s'ha vist en els capítols previs, un metamodel defineix una sintaxi abstracta, i els models que la "respecten" es diu que són conformes a aquesta. Quan es defineix una transformació entre models, es fa respecte dels seus metamodels. Posteriorment s'utilitzarà un motor de transformació per a executar-la sobre models concrets. En general, una transformació pot tenir diversos models origen i destinació (*source model* i *target model*, en anglès, respectivament), i és possible a més que el metamodel origen i destinació siguin el mateix. Un exemple de transformació en què coincideixen el model origen i destinació el trobem, per exemple, quan es proporcionen transformacions que descriuen casos d'estudi d'un sistema.

Figura 24. Participants d'una transformació de models



Generalment, una transformació de models està formada per un conjunt de **regles de transformació**, que són considerades com les unitats de transformació més petites. Cadascuna de les regles descriu com un o més elements del model origen són transformats en un o més elements del model destinació.

Nota

En parlar d'"elements" ens referim a classes, atributs, relacions, etc.

4.3. Tipus de transformacions

En MDE es descriuen diferents tipus de transformacions entre models, depenent de diversos criteris:

a) Nivell d'abstracció dels models d'entrada i sortida:

- Transformacions **verticals**. Relacionen models del sistema situats en diferents nivells d'abstracció i es poden aplicar tant en sentit descendent[14] com en sentit ascendent[15], i aquests últims són el resultat d'aplicar un procés d'enginyeria inversa.

- Transformacions **horitzontals**. Relacionen els models que descriuen un sistema des d'un nivell d'abstracció similar. S'utilitzen també per a man-

Consulta recomanada

En el treball següent de Krzysztof Czarnecki i Simon Helen (2006) es fa una classificació molt detallada i completa de les transformacions de models: "Feature-Based Survey of Model Transformation Approaches", IBM System Journal (vol. 45, núm. 3, pàg. 621-645).

(14)Transformacions de PIM a PSM, també denominades *refinaments*.

tenir la consistència entre diferents models d'un sistema, és a dir, garanteixen que la informació modelitzada sobre una entitat en un model és consistent amb el que es diu sobre aquesta entitat en qualsevol altra especificació situada al mateix nivell d'abstracció.

[15] De PSM a PIM, anomenades *abstraccions*.

b) Tipus de llenguatge que s'utilitza per a especificar les regles. Es distingeix entre llenguatges **declaratius**, **imperatius** o **híbrids** (si permeten tots dos tipus de manera combinada)

Consulta recomanada

L'article següent de Perdita Stevens (2010) estudia l'especificació de transformacions bidireccionals utilitzant QVT: "Bidirectional Model Transformations in QVT: Semantic Issues and Open Questions" *SoSyM* (vol. 9, núm. 1, pàg. 7-20).

c) Atenent a la direccionalitat, ens trobem amb:

- Transformacions **unidireccionals**, les regles s'executen en una sola direcció.

- Transformacions **bidireccionals**. En aquestes, les transformacions es poden aplicar en totes dues direccions[16].

[16] Del model origen a la destinació i de la destinació a l'origen.

d) Depenent dels models origen i destinació:

- Transformacions **exògenes**. Els metamodels origen i destinació són diferents. És el cas més comú: es té un model d'entrada i s'obté el de sortida a partir d'aquest aplicant les regles de la transformació.

- Transformacions **endògenes**. Aquí, els metamodels origen i destinació són el mateix, es tracta de les anomenades transformacions *in-place*: les regles de transformació es van aplicant sobre el model d'entrada mateix, de manera que aquest es va transformant fins que no es pot aplicar cap regla més i el model d'entrada passa a ser el de sortida.

e) Atenent al tipus de model destinació:

- Transformacions de **model a model** (M2M), les quals generen models a partir d'altres models. Es poden classificar al seu torn en subgrups, com es veurà en l'apartat 4.4.

- Transformacions de **model a text** (M2T), que generen cadenes de text a partir de models. Són usades, per exemple, per a generar codi i documents. Es classifiquen al seu torn en subgrups, com s'explica en l'apartat 4.5.

4.4. Llenguatges de transformació de model a model (M2M)

L'OMG identifica, per mitjà d'MDA, quatre tipus de transformacions de model a model (M2M) dins del cicle de vida del programari:

1) Transformacions PIM2PIM. Aquestes transformacions s'usen en el tractament de models independents de la plataforma i s'apliquen quan s'obtenen, filtren o especialitzen uns PIM a partir d'altres.

2) Transformacions PIM2PSM. Transformen un model independent de la plataforma en el model corresponent en una plataforma concreta.

3) Transformacions PSM2PSM. S'utilitzen per a refinar un model desplegat en una plataforma concreta.

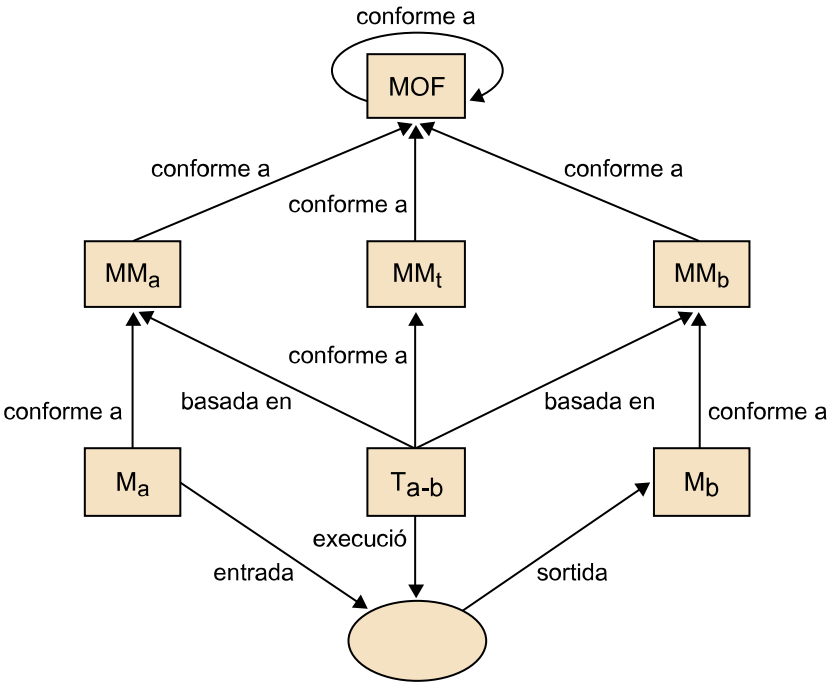
4) Transformacions PSM2PIM. Abstreuen models desplegats en plataformes concretes en models independents de la plataforma.

Nota

Les transformacions PSM2PIM s'utilitzen, sobretot, per a tasques d'enginyeria inversa i de modernització de sistemes.

De tots els llenguatges existents per a descriure transformacions de model a model, els més usats avui dia són QVT (*query/view/transformation*) i ATL (*Atlas transformation language*). Tots dos presenten el mateix context operacional, que es mostra en la figura 25.

Figura 25. Context operacional d'ATL i QVT



La transformació està representada per $T_{a\text{-}b}$, l'execució de la qual resulta en la creació d'un model M_b a partir d'un model M_a. Les tres entitats esmentades ($T_{a\text{-}b}$, M_b i M_a) són models conformes als metamodels MOF MM_t, MM_b i MM_a, respectivament. MM_a i MM_b representen les sintaxis abstractes dels llenguatges origen i destinació de la transformació, mentre que MM_t representa la sintaxi abstracta del llenguatge de transformació usat, en aquest cas QVT o ATL.

Nota

Recordem que, com es va veure en l'apartat "Conceptes bàsics", MOF (*meta-object facility*) és el llenguatge de l'OMG per a descriure metamodels.

En els apartats següents es detallen QVT i ATL per separat, recalcant especialment el segon, amb el qual farem alguns exemples.

4.4.1. QVT: *query-view-transformation*

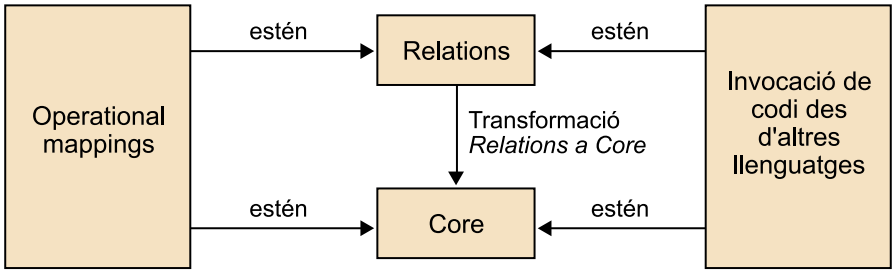
QVT (de l'anglès *query-view-transformation*) és un estàndard proposat per l'OMG i format per un conjunt de llenguatges destinats a les transformacions de models. És capaç d'expressar transformacions, vistes i peticions entre models dins del context de l'arquitectura de metamodelització MOF 2.0.

Web recomanat

L'última especificació de MOF-QVT (gener de 2011) es troba a http://www.omg.org/spec/qvt/1.1/.

La sintaxi abstracta de QVT es defineix com un metamodel MOF 2.0. Aquest metamodel defineix tres subllenguatges per a transformar models, els quals componen un llenguatge de transformacions **híbrid** amb constructors tant declaratius com imperatius. Aquests llenguatges es diuen *Relations*, *Core* i *Operational Mappings*, i s'organitzen en una estructura per nivells que es mostra en la figura 26. La definició d'aquests tres llenguatges incorpora en la seva sintaxi el llenguatge estàndard OCL per a expressar consultes (*queries*) sobre els models. Els llenguatges *Relations* i *Core* són declaratius i es troben en dos nivells d'abstracció diferents. El tercer llenguatge, *Operational Mappings*, és un llenguatge imperatiu que estén als altres dos llenguatges. A continuació detallem cadascun d'aquests llenguatges.

Figura 26. Arquitectura en nivells dels llenguatges de QVT



Consulta recomanada

El treball següent de Frédéric Jouault i Ivan Kurtev (2006) compara els llenguatges ATL i QVT i estudia la interoperabilitat entre tots dos: "On the Architectural Alignment of ATL and QVT". SAC 2006 (pàg. 1188-1195).

1) *Relations*. Aquest llenguatge permet especificar transformacions com un conjunt de relacions entre models. Això permet al desenvolupador crear elements en el model destinació a partir d'elements del model origen, i també aplicar canvis sobre models existents. Aquest llenguatge s'encarrega de la manipulació automàtica de les traces (estructures de dades en què es desen les relacions entre els conceptes dels models origen i els models destinació) de manera transparent al desenvolupador.

2) *Core*. Aquest llenguatge declaratiu és més simple que l'anterior. Per tant, les transformacions escrites en *Core* solen ser més llargues que les seves equivalents especificades en *Relations*. Les traces en *Core* són tractades com a elements del model, és a dir, el seu maneig no és transparent en aquest cas, per la qual cosa el desenvolupador es responsabilitza de crear i usar les traces. Una de les raons de ser d'aquest llenguatge és proporcionar una base sobre la qual es pugui especificar la semàntica del llenguatge *Relations*. Aquesta semàntica es dóna com una transformació des de *Relations* a *Core*, la qual és escrita en el llenguatge *Relations*.

De vegades és difícil desenvolupar una solució purament declarativa per a una transformació determinada. Per això, QVT proposa dos mecanismes per a "estendre" els llenguatges declaratius *Relations* i *Core* amb nous conceptes i mecanismes: el llenguatge *Operational Mappings* i un nou mecanisme per a invocar la funcionalitat de transformacions implementades en altres llenguatges (representat en la figura 27 per el rectangle de l'extrem dret).

3) *Operational Mappings*. Com s'ha esmentat, aquest llenguatge estén el llenguatge *Relations* amb constructors imperatius i constructors OCL. Així, aquest llenguatge és purament imperatiu i similar als llenguatges procedimentals de programació tradicional. A més, incorpora elements constructius dissenyats per a operar (crear, modificar i eliminar) sobre el contingut dels models. *Operational Mappings* s'estructura bàsicament en procediments o operacions, denominats *mapping operations* i *helpers* o *queries*.

4) Invocació de codi des d'altres llenguatges (*Black Box* en anglès). Aquest mecanisme permet introduir i executar codi extern en la transformació en temps d'execució, la qual cosa permet implementar algorismes complexos en qualsevol llenguatge de programació i reutilitzar biblioteques existents. No obstant això, això fa que algunes parts de la transformació siguin opaques, la qual cosa crea un risc potencial, ja que la funcionalitat serà arbitrària i no podrà ser controlada pel motor d'execució.

Nota

En aquest mòdul no entrarem detalladament sobre els mecanismes de QVT *Operational Mappings* i *Black Box*, per ser d'un nivell més avançat. El lector interessat pot trobar exemples en l'enllaç següent: http://www.eclipse.org/m2m/qvto/doc/M2M-QVTO.pdf

4.4.2. ATL: *Atlas transformation language*

Web recomanat

El manual i nombrosos exemples d'ATL es poden trobar a http://www.eclipse.org/m2m/atl/doc/

ATL és un llenguatge de transformació de models desenvolupat sobre EMF (*Eclipse modeling framework*) com a part de la plataforma AMMA (*Atlas model management architecture*). Els desenvolupadors d'ATL es van inspirar en QVT per a crear-lo, i, com en aquest, OCL forma part del llenguatge.

Web recomanat

La descripció d'AMMA es troba detallada a http://wiki.eclipse.org/AMMA

A pesar que ATL és un llenguatge híbrid, que proporciona tant constructors declaratius com imperatius, és recomanat usar de manera declarativa tret que sigui estrictament necessari, és a dir, que la transformació no es pugui especificar d'una altra manera.

Les transformacions escrites amb ATL són unidireccionals, les quals operen sobre models origen només de lectura i generen models destinació només d'escriptura. Això vol dir que les transformacions no poden navegar (consultar) el model de sortida que es va creant, ni tampoc modificar el model d'entrada (encara que sí navegar-hi). Si es vol implementar una transformació bidireccional amb ATL, és necessari definir una transformació en cada sentit.

El codi de qualsevol transformació ATL s'organitza en tres parts ben definides: la **capçalera**, un conjunt de ***helpers*** (opcional) i un conjunt de **regles**.

En **la capçalera** es declara informació general com, per exemple, el nom del mòdul (és a dir, el nom de la transformació), els metamodels d'entrada i sortida i la importació de biblioteques. Els ***helpers*** són subrutines que s'usen per a evitar codi redundant: es poden veure com els equivalents als mètodes en Java. Les **regles** són la part més important de les transformacions ATL, ja que descriuen com es transformen els elements del model origen en els elements del model destinació. Estan formades per vincles (*bindings*), de manera que cadascuna expressa una correspondència (*mapping*) entre un element d'entrada i un de sortida. A continuació es mostren tres tipus de regles existents en ATL i les seves propietats: *ATL matched rules*, *ATL lazy rules* i *ATL unique lazy rules*.

Vegeu també

En l'apartat 4.4.4. es mostra un cas d'estudi en què s'exemplifica l'ús de les regles ATL.

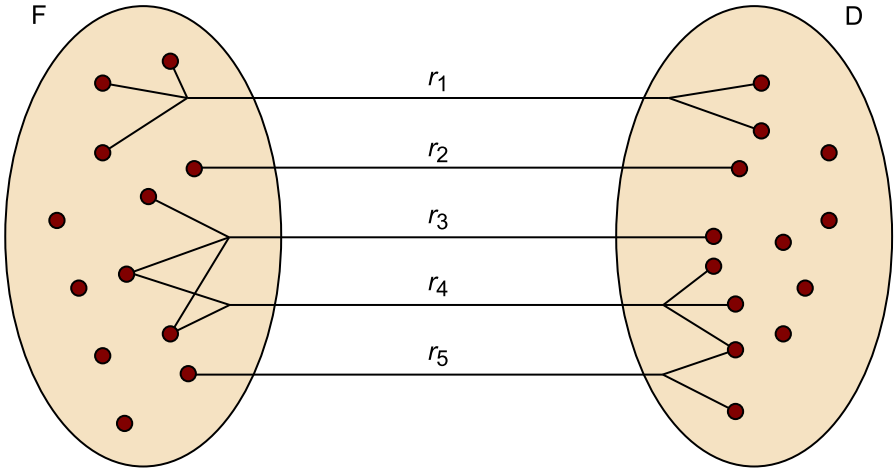
ATL matched rules

En una transformació ATL tindrem regles que s'aplicaran sobre els elements del model origen per a generar elements en el model destinació. Les regles declaratives que fan aquesta acció s'anomenen *matched rules*. Aquestes regles constitueixen les peces clau de les transformacions ATL.

Una *matched rule* s'identifica pel nom que se li dóna, té com a entrada un o més tipus d'objectes del model origen i genera un o més tipus d'objectes del model destinació especificant els valors que reben els atributs i referències dels objectes creats en el model destinació.

Aquestes regles són declaratives, encara que també admeten una part imperativa opcional, la qual s'executa una vegada que ha acabat la part declarativa.

Figura 27. Correspondències entre els models origen i destinació definides en *matched rules*



ATL lazy rules

ATL també ofereix la possibilitat d'usar un altre tipus de regles declaratives, anomenades *lazy rules* i *unique lazy rules*.

Les *lazy rules* són com les *matched rules* amb la diferència que són executades només quan són cridades des d'una altra regla (les *matched rules* no se les invoca des de cap lloc).

ATL unique lazy rules

ATL ofereix un tercer tipus de regla declarativa anomenada *unique lazy rule*. Aquestes regles són un tipus especial de les *lazy rules* que sempre retornen el mateix element destinació per al mateix element origen. Les *lazy rules*, en canvi, sempre creen nous elements del model destinació en les seves crides.

Blocs imperatius

Com es va esmentar anteriorment, ATL proporciona **constructors imperatius** que s'han d'usar quan la transformació és massa complexa per a ser expressada declarativament. La part imperativa la componen les *called rules* i els *action blocks*. Els *action blocks* són seqüències de sentències imperatives que constitueixen la part imperativa de les regles declaratives. Es defineixen i executen després de la part declarativa i es poden usar en comptes d'aquesta o en combinació amb un element destinació creat en la part declarativa. En aquesta part imperativa es permeten fer assignacions, utilitzar condicions (amb *if*), definir bucles (amb *for*) i invocar *called rules*. Les *called rules* són regles que es criden des de la part imperativa d'altres regles, per la qual cosa serveixen de procediments. Així, aquestes regles es poden veure com un tipus especial de *helpers*, ja que han de ser cridades explícitament per a ser executades i accepten paràmetres d'entrada. No obstant això, al contrari dels *helpers*, aquestes regles poden generar elements en el model destinació. Aquestes regles s'han de cridar des de la part imperativa d'altres regles.

4.4.3. Diferents enfocaments per a les transformacions M2M

Web recomanat

Més informació sobre JMI en http://java.sun.com/products/jmi/

Segons la manera com s'executen les transformacions de model a model, hi ha diversos mecanismes per a especificar-les. A causa de la gran oferta que hi ha, és important que els desenvolupadors siguin capaços de comparar i triar les eines i llenguatges més apropiats per al seu problema concret. De fet, per a l'especificació d'algunes transformacions complexes entre models pot ser convenient utilitzar diversos llenguatges. Anteriorment hem descrit els dos llenguatges de transformació de models més importants avui dia: QVT i ATL. A continuació esmentem altres enfocaments existents actualment:

- **Enfocaments de manipulació directa**. Són els de més baix nivell. Aquest mecanisme proporciona una infraestructura mínima per a organitzar les transformacions, per la qual cosa els usuaris normalment han

d'implementar les seves regles de transformació des de zero i amb llenguatges de programació orientats a objectes com Java.

Un exemple d'aquests llenguatges és JMI (*Java metadata interface*).

- **Enfocaments dirigits a estructures**. En fer una transformació de models amb aquests llenguatges, se segueixen dues fases: en la primera es crea l'estructura jeràrquica del model destinació, en la segona s'estableixen els atributs i referències del model.

  Un exemple d'aquest enfocament és el *framework* per a fer transformacions M2M ofert per OptimalJ.

- **Enfocaments operacionals**. Aquesta categoria engloba llenguatges que ofereixen un enfocament similar a la manipulació directa però que ofereixen un suport més refinat en escriure les transformacions. Així per exemple, poden oferir facilitats especials com el traçat per mitjà de biblioteques.

  Exemples de llenguatges d'aquest tipus són *QVT Operational Mappings*, *XMF-Mosaic's executable MOF*, MTL i Kermeta.

- **Enfocaments basats en plantilles**. Aquestes plantilles són models que allotgen metacodi escrit en la sintaxi concreta del llenguatge destinació. Això ajuda al desenvolupador a predir el resultat d'una instanciació de la plantilla. El desenvolupador es pot guiar per la plantilla i completar el que manca.

- **Enfocaments relacionals**. En aquesta categoria hi ha aquells llenguatges declaratius en què el concepte principal són les relacions matemàtiques. La idea és especificar les relacions entre els elements origen i destinació usant restriccions.

  Exemples de llenguatges d'aquest tipus són *QVT Relations*, MTF, *Kent model transformation language*, Tefkat i AMW.

- **Llenguatges basats en transformacions de grafs**. Aquests llenguatges operen sobre grafs tipats, etiquetats i amb atributs, que no són més que representacions en forma de graf de models de classes simplificats. En aquests llenguatges, un model es representa com un graf en què els objectes són nodes i els enllaços entre aquests són arestes. Cada regla descriu un patró en forma de subgraf (part esquerra de la regla) que, cada vegada que apareix en el model (*matching*), s'ha de substituir per un altre patró (part dreta de la regla). Aquest tipus de transformacions s'anomena *in-place*, ja que és el graf d'entrada el que es va modificant a mesura que s'apliquen les regles de transformació fins que no hi hagi cap *matching* de cap regla amb el graf. En aquest moment, el resultat de la transformació és el graf mateix.

Entre els llenguatges d'aquest tipus podem trobar AGG, AToM^3, VIATRA, GReAT, UMLX, BOTL, MOLA i Fujaba.

Vegeu també

Els llenguatges de transformació de grafs es van esmentar en l'apartat "Semàntica", perquè també permeten definir la semàntica dels llenguatges específics de domini.

- **Llenguatges híbrids**. Aquests llenguatges combinen diferents tècniques de les mostrades anteriorment, bé com a components separats o bé d'una manera més fina al nivell de les regles.

Un exemple d'aquest tipus de llenguatges és QVT, que es compon dels tres llenguatges explicats en l'apartat 4.4.1: *Relations*, *Operational Mappings* i *Core*. ATL és un altre exemple de llenguatge híbrid (admet barrejar blocs declaratius i imperatius en les seves regles).

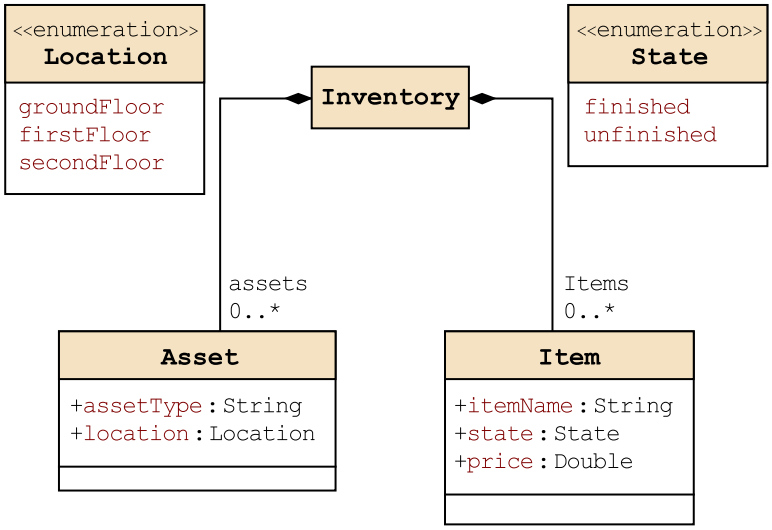
Nota

Alguns termes, com *matching* o *in-place*, no se solen traduir de l'anglès perquè han estat adoptats tal qual. Així no se sol utilitzar emparellament per a referir-se a *matching*, o *in situ* per a transformacions *in-place*.

4.4.4. Cas d'estudi de transformació M2M (ATL)

En l'apartat "Introducció a MDA i MDE" es va estudiar la modelització d'una cadena de muntatge, i el seu metamodel és el mostrat en la figura 7. Ara el nostre objectiu és obtenir vistes dels models de la cadena de muntatge. Aquestes noves vistes seran models d'inventari utilitzats en l'empresa, i el seu metamodel es mostra en la figura 28.

Figura 28. Metamodel de l'inventari



Amb aquest metamodel es poden definir models que ofereixin informació sobre l'inventari d'una cadena de muntatge en un moment determinat. Així, un inventari es compon dels recursos inventariables de la cadena de muntatge (classe *Asset*) i les peces que es tinguin a la fàbrica en el moment que es fa l'inventari, és a dir, els elements fungibles (classe *Item*). Els objectes de tipus *Asset* són d'un cert tipus segons siguin un tipus de màquina (subclasse de *Machine*) o una safata (*Tray*) en el model origen, i registren informació sobre la seva ubicació (valors planta baixa, primera o segona) d'acord a un tipus enumerat *Location*. Els objectes de tipus *Item* tenen un nom, segons la seva classe (subclasse de *Part* a la qual pertanyen), un estat (acabat o no) especificat per un enumerat *State* i el seu preu.

Centrem-nos en el model de la cadena de muntatge de la figura 9. Aquest model consta de quatre màquines (dos generadors, una acobladora i una polidora), quatre safates i sis peces (dos caps de martell, un mànec de martell,

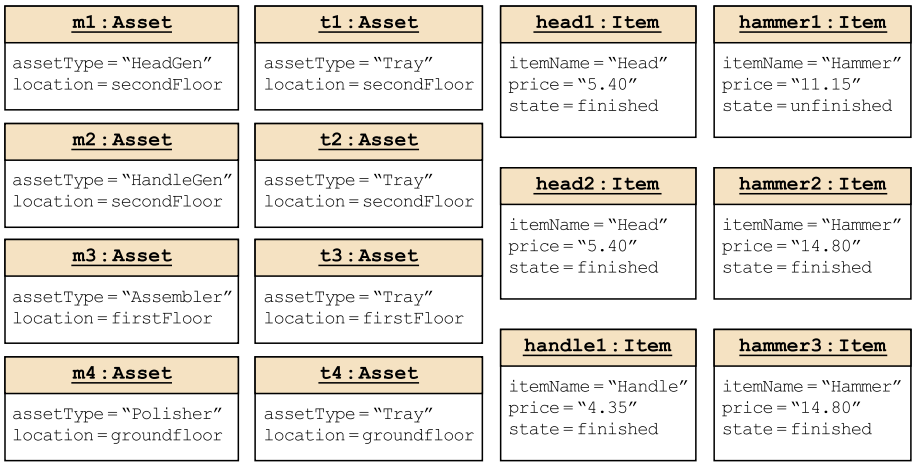
un martell no polit i dos martells polits). En el model no es va especificar res sobre la posició dels elements, però per a la transformació suposarem que sí que tenim aquesta informació sobre el model i que aquesta informació és la mostrada en la taula 1. La taula reflecteix la posició (coordenades *x* i *y*) que suposem per a cadascuna de les màquines i safates del model.

Taula 1. Posicions per als elements del model de la figura 8

pos	*m1*	*m2*	*t1*	*t2*	*m3*	*t3*	*m4*	*t4*
x	6	6	6	6	4	4	1	1
y	1	4	2	5	3	4	5	7

Considerant aquesta informació, i prenent com a model d'entrada de la transformació el model de la figura 9, el model de sortida que volem obtenir amb la nostra transformació *Plant2Inventory* és el mostrat en la figura 29. Explicarem detalladament quins són els criteris per a associar un pis (*location*), preu (*price*) i estat (*state*) a cada element del model en detallar com funciona la transformació.

Figura 29. Model d'inventari resultat de la transformació *Plant2Inventory*



Ara descriurem la transformació ATL que s'encarrega de dur a terme aquesta transformació.

En **la capçalera** de la definició de la nostra transformació declarem el nom del mòdul i els metamodels d'entrada i sortida:

```
module Plant2Inventory; -- Module Template
create OUT : Inventory  from IN : Plant;
```

Nota

Recordem que tota transformació ATL està formada per **una capçalera** i una sèrie de **regles**.

D'altra banda, les **regles** que transformaran els elements del model origen en els elements del model destinació en el nostre exemple són les següents:

1) Regla **Machine2Asset**: per a cada instància de la classe *Machine*, crea una instància de la classe *Asset*.

- L'`assetType` contindrà el nom de la subclasse concreta de `Machine`. Per exemple '`HeadGen`', '`HandleGen`', '`Assembler`', '`Polisher`'.

- L'atribut `location` indica si la màquina és a la planta baixa, primera o segona. Seran a la planta baixa aquelles màquines amb coordenada `y` que valgui 0 o 1, a la primera planta aquelles les que tinguin coordenada `y` entre 2 i 4 i a la segona planta aquelles que tinguin un valor entre 5 i 7 per a aquesta coordenada.

2) Regla **Tray2Asset**: per a cada instància de la classe `Tray`, crea una instància de la classe `Asset`.

- L'`assetType` contindrà el nom de la classe, és a dir, `'Tray'`.

- L'atribut `location` indica si la màquina és a la planta baixa, primera o segona. Es calcula com s'ha descrit en la regla anterior.

3) Regla **Part2Item**: per a cada instància de la classe `Part` crea una instància de la classe `Item`.

- L'atribut `itemName` contindrà el nom de la subclasse concreta de `Part`, és a dir, '`Head`', '`Handle`' o '`Hammer`'.

- L'atribut `state` indica si la peça està acabada o no. Es considera que els martells que no han estat polits no estan acabats. Per contra, els martells polits, els caps de martell i els mànecs de martell sí que estan acabats, ja que poden ser reutilitzats per a qualsevol altra finalitat.

- `Price` indica el preu de la peça. Els caps de martell tenen un preu de 5,40 €, els mànecs costen 4,35 €, un martell sense polir 11,15 € i un de polit 14,80 €.

4.4.5. *ATL matched rules*

Per a mostrar l'ús de les *matched rules*, l'extracte següent de codi correspon a la primera de les regles descrites en l'apartat anterior:

```
rule Machine2Asset {
   from  m : Plant!Machine
   to    a : Inventory!Asset(
              assetType <- m.getMachineType(),
              location <- m.y.getFloor()
           )
}
```

La regla es diu `Machine2Asset`, i tindrà d'entrada una màquina i generarà un actiu. Els elements d'entrada s'especifiquen després de la paraula reservada `from`, i els de sortida després de la paraula reservada `to`. Quan es vol fer referència a una classe s'escriu el nom del metamodel (aquest nom s'assigna en la capçalera de la transformació) seguit de "!" i del nom de la classe. Així, per a referir-se a la classe `Machine` cal escriure `Plant!Machine`, i per a referir-se a la classe `Asset` cal escriure `Inventory!Asset`. En la generació de l'element de sortida veiem com s'inicialitzen els seus atributs. En aquest cas, els dos atributs d'`Asset`, `assetType` i `location`, s'han inicialitzat fent ús de dos `helpers`:

```
helper context Plant!Machine def : getMachineType() : String =
   if (self.oclIsTypeOf(Plant!HeadGen))
      then 'HeadGen'
      else if (self.oclIsTypeOf(Plant!HandleGen))
         then 'HandleGen'
         else if (self.oclIsTypeOf(Plant!Assembler))
            then 'Assembler'
            else 'Polisher'
         endif
      endif
   endif ;

helper context Integer def : getFloor() : Inventory!Location =
   if (self >= 0 and self <= 1)
     then #groundFloor
     else if (self >= 2 and self <= 4)
       then #firstFloor
       else #secondFloor
    endif
   endif ;
```

Com es va esmentar anteriorment, els *helpers* ens permeten definir procediments i funcions. Permeten definir codi ATL, que es pot cridar des de diferents parts de la transformació. Un *helper* es defineix amb els elements següents:

- El seu nom, utilitzat per a invocar-lo.
- Un tipus de context (opcional). Especifica el context en el qual es defineix l'atribut. Un `self` que s'utilitza dins del *helper* es refereix al context.
- Valor per retornar. En ATL, tot *helper* ha de retornar un valor.
- Una expressió ATL que representa el codi del *helper*.

- Un conjunt de paràmetres d'entrada (opcional). Igual que en Java, un paràmetre d'entrada és identificat per una parella (nom del paràmetre, tipus del paràmetre).

El primer dels *helpers* anteriors, `getMachineType`, rep un objecte de tipus màquina com a entrada i retorna una cadena de text que es correspon amb el nom de la subclasse concreta de la màquina. Com ATL utilitza OCL, en aquest *helper* s'ha fet ús del mètode `oclIsTypeOf` que es va explicar en l'apartat 2. El *helper* comprova a quina subclasse no abstracta de `Machine` pertany l'objecte des del qual es diu al *helper* i retorna el nom de la classe com a cadena de text.

El segon *helper*, `getFloor`, rep un enter que es correspon amb la coordenada `y` d'un objecte i retorna la ubicació a la qual correspon aquesta coordenada. El valor per retornar és del tipus enumerat `Location`. Podem observar en el *helper* que per a especificar els valors d'un tipus enumerat s'utilitza el caracter '#' seguit del nom del literal.

La segona de les regles especificades anteriorment (`Tray2Asset`) es correspon amb la *matched rule* següent:

```
rule Tray2Asset {
   from  t : Plant!Tray
   to    a : Inventory!Asset(
           assetType <- 'Tray',
           location <- t.y.getFloor()
         )
}
```

En aquesta regla també s'utilitza el *helper* `getFloor` per a obtenir la ubicació de la safata. Aquesta regla és fàcilment entesa després de l'explicació de l'anterior. La tercera regla (`Part2Item`) es mostra a continuació:

```
rule Part2Item {
   from  p : Plant!Part
   to    i : Inventory!Item(
           itemName <- p.getItemName(),
           state <- p.getPartState(),
           price <- p.getPrice()
         )
}
```

Aquesta regla converteix els objectes de tipus `Part` del model origen en objectes de tipus `Item` en el model destinació. Observem que els valors dels atributs de l'element destinació s'han inicialitzat utilitzant tres *helpers*. El primer, `getItemName`, és molt similar al *helper* `getMachineType` mostrat anteriorment. El *helper* `getPartState` és:

Exercici proposat

Definir el *helper* `getItemName`.

```
helper context Plant!Part def : getPartState() : Inventory!State =
   if self.oclIsTypeOf(Plant!Head) or self.oclIsTypeOf(Plant!Handle)
     then #finished
     else if  (self.oclIsTypeOf(Plant!Hammer) and self.isPolished)
      then #finished else #unfinished
     endif
   endif ;
```

Aquest *helper* assigna l'estat `finished` a aquelles peces que siguin caps o mànecs. Per als martells, comprova si està polit (per a assignar-li `finished`) o no (per a assignar-li `unfinished`). El tercer *helper*, `getPrice`, assigna el preu a cada peça com es va explicar anteriorment.

Exercici proposat

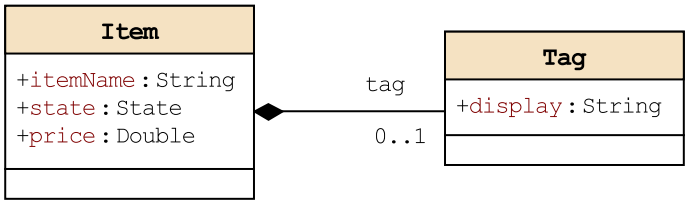
Definir el *helper* `getPrice` la capçalera del qual és:

```
helper context Plant!Part def : getPrice() : Real.
```

4.4.6. *ATL lazy rules*

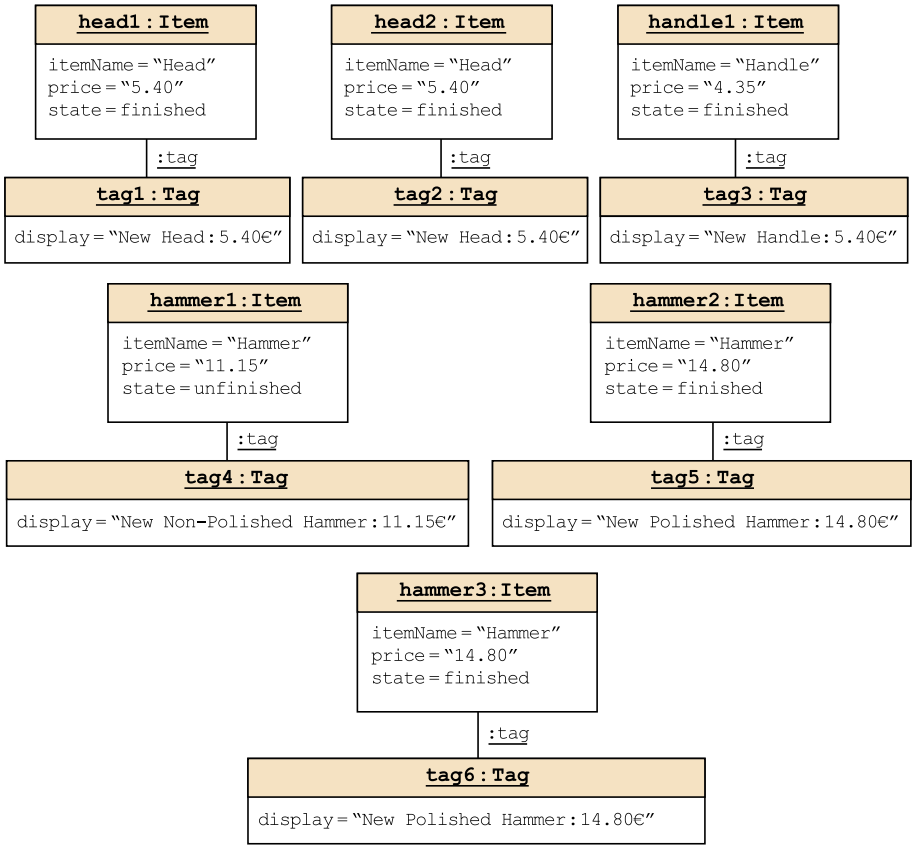
Per a mostrar l'ús d'aquestes regles, suposem que afegim una classe addicional `Tag` al metamodel d'inventaris, tal com mostra la figura 30.

Figura 30. Extensió del metamodel d'inventaris



L'objectiu de la nova classe `Tag` és etiquetar els articles per a poder vendre'ls. Així, sempre que es crea una instància de tipus `Item` mitjançant la transformació, ara també se'n voldrà tenir una etiqueta.

Figura 31. Model d'inventari resultat de la transformació `Plant2InventoryWithTags` (els objectes de tipus `Asset` no es mostren)



Cridem la nova transformació `Plant2InventoryWithTags`. El model de sortida que ha de quedar ara (figura 31) és una extensió del model de la figura 28 en el qual apareixen objectes de tipus `Tag`: per a crear els objectes etiqueta podem utilitzar una *lazy rule*.

```
rule Part2Item2 {
   from  p : Plant!Part
   to    i : Inventory!Item(
               itemName <- p.getItemName(),
               state <- p.getPartState(),
               price <- p.getPrice(),
               tag <- thisModule.CreateTag(p)
             )
}
lazy rule CreateTag {
   from  p : Plant!Part
   to    t : Inventory!Tag(
               display <- p.getDisplay()
            )
}
```

La regla `Part2Item2` és la mateixa *matched rule* que anteriorment, encara que ara cal donar-li un valor a la nova referència, `Tag`, quan es crea una instància de la classe `Item`. Per a crear la nova instància de tipus `Tag` cridem una *lazy rule* anomenada `CreateTag` i li passem l'objecte de tipus `Part` que va disparar la *matched rule*. Per a invocar una *lazy rule* cal escriure "`thisModule.`" davant del seu nom. Els objectes d'entrada de la *lazy rule*, que apareixen seguits de la

paraula reservada `from`, són els que es passen com a paràmetre. Aquesta *lazy rule* crea un objecte de tipus `Tag` i inicialitza el seu únic atribut mitjançant un *helper*.

Exercici proposat

Definiu el *helper* `getDisplay()` i deduïu-ne comportament observant el model de la figura 28.

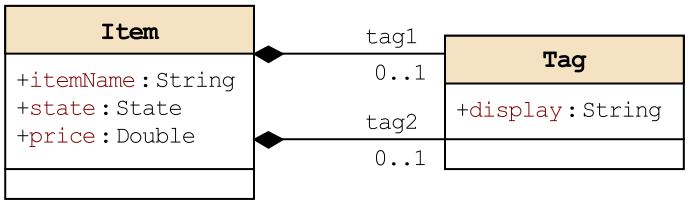
El codi anterior, en el qual s'ha utilitzat una *lazy rule*, es podria haver escrit, alternativament, solament amb la *matched rule* i estenent-la perquè ella mateixa creés els dos elements destinació (l'`Item` i el `Tag`):

```
rule Part2Item {
   from  p : Plant!Part
   to    i : Inventory!Item(
            itemName <- p.getItemName(),
            state <- p.getPartState(),
            price <- p.getPrice(),
            tag <- t
          ),
         t : Inventory!Tag(
            display <- p.getDisplay()
          )
}
```

4.4.7. *ATL unique lazy rules*

Ara explicarem aquest tipus de regles amb un exemple. Imaginem que en el metamodel d'inventaris els articles poden tenir ara dues etiquetes (iguals), i que les modelitzem amb relacions `tag1` i `tag2`:

Figura 32. Extensió del metamodel d'inventaris

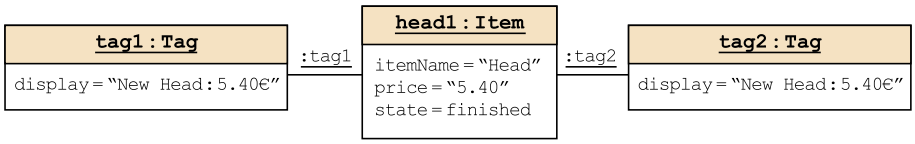


Ara, en la regla `Part2Item` hem de tenir en compte les dues relacions. El codi que resulta d'utilitzar la *lazy rule* anterior pot ser el següent:

```
rule Part2Item3 {
   from  p : Plant!Part
   to    i : Inventory!Item(
               itemName <- p.getItemName(),
               state <- p.getPartState(),
               price <- p.getPrice(),
               tag1 <- thisModule.CreateTag(p),
               tag2 <- thisModule.CreateTag(p)
             )
}
lazy rule CreateTag {
   from  p : Plant!Part
   to    t : Inventory!Tag(
               display <- p.getDisplay()
             )
}
```

Observem que s'ha cridat la mateixa *lazy rule* dues vegades amb el mateix element d'entrada. Si ens fixem només en l'objecte `head1` de la figura 33, ara té dues etiquetes associades. Totes dues tenen el mateix contingut però són **dues instàncies diferents** de la classe `Tag`:

Figura 33. Creació de dues instàncies de la classe `Tag` invocant dues vegades la mateixa *lazy rule*

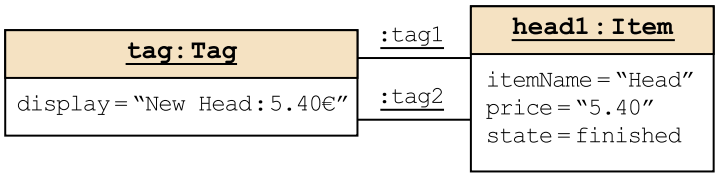


Si la regla `CreateTag`, que ara mateix és una *lazy rule*, la reemplacem per una *unique lazy rule*, s'obté un resultat diferent. Sintàcticament, una *unique lazy rule* es diferencia d'una *lazy rule* només en la definició, que inclou la paraula `unique`:

```
unique lazy rule CreateTag {
   from  p : Plant!Part
   to    t : Inventory!Tag(
         display <- p.getDisplay()
        )
}
```

Ara, la primera invocació de la regla `CreateTag` des de la regla `Part2Item` crea un objecte de tipus `Tag` i aquest és associat amb l'objecte de tipus `Item` mitjançant la relació `tag1`. En la segona crida a `CreateTag`, que ara és una *unique lazy rule*, el motor d'execució d'ATL recupera la instància que es va crear en la crida anterior, ja que el paràmetre d'entrada és el mateix, en comptes de crear un nou objecte. D'aquesta manera, la relació `tag2` associa el mateix `Item` amb el mateix *Tag* que la relació `tag1`.

Figura 34. Model corresponent al de la figura 33 però usant una *unique lazy rule* en comptes de *lazy rule*



4.4.8. Blocs imperatius

Ara veurem un exemple molt simple sobre com s'escriu la part imperativa d'una regla. L'ús que donem a la part imperativa en aquest exemple és el de modificar un objecte que s'ha creat en la part declarativa de la mateixa regla. Recordem la regla `Part2Item` vista anteriorment. En aquesta, es crea un objecte `Item` i un altre `Tag` a partir d'un objecte de tipus *Part*. En la part declarativa s'assigna un nom a l'objecte de tipus `Item` (en el seu atribut `itemName`) mitjançant la crida al *helper* `getItemName`. Imaginem que volem afegir quelcom més a aquest nom, i per a això usarem la part imperativa de la regla.

Un exemple és el següent:

```
rule Part2Item {
   from  p : Plant!Part
   to    i : Inventory!Item(
                 itemName <- p.getItemName(),
               …
             )
   do {
              i.itemName <- i.itemName + ' imperative code';
   }
}
```

En executar aquesta regla, el camp `itemName` d'un objecte de tipus martell serà "`Hammer imperative code`" en comptes de simplement "`Hammer`". Cal destacar que aquest exemple senzill simplement mostra la manera com s'escriu codi en la part imperativa d'ATL i com s'accedeix als objectes creats en la part declarativa i als seus atributs. En aquest exemple la cadena de caràcters que s'afegeix es podria haver afegit en la part declarativa.

4.5. Llenguatges de transformació de model a text (M2T)

Les transformacions de model a text (M2T) són aquelles que prenen un model com a entrada i retornen una cadena de text com a sortida. Hi ha dos enfocaments a l'hora d'especificar aquestes transformacions:

- Enfocaments basats en **visites**. Consistent a proporcionar un mecanisme per a recórrer la representació interna d'un model i escriure text en un flux de text.

  Un exemple és Jamda, un *framework* orientat a objectes que proporciona un conjunt de classes per a representar models UML, una API per a manipular models i un mecanisme de visita per a generar codi.

Web recomanat

Més informació sobre Jamda en The Jamda Project: http://jamda.sourceforge.net

- Enfocaments basats en **plantilles**. La majoria de les eines MDA disponibles actualment suporten la generació de codi amb aquest mecanisme: openArchitectureWare, JET, FUUT-je, Codagen Architect, AndroMDA, ArcStyler, MetaEdit+, OptimalJ, TCS, MOFScript. Una plantilla consisteix a donar el text destinació, i conté fragments de metacodi per a accedir a informació del model origen i permetre l'expansió iterativa del text.

Entre les eines existents destaquem **MOFScript**, una eina i llenguatge per a declarar transformacions M2T. Proporciona un llenguatge independent del metamodel que permet usar qualsevol tipus de metamodel i les seves instàncies per a la generació de codi. També pot ser utilitzat per a definir transformacions de model a model quan es vol construir el model des de zero o per a augmentar models existents. Aquesta eina està basada en EMF i Ecore com a plataforma de metamodelització. El segon llenguatge que destaquem és **TCS**, que es descriu en l'apartat següent.

Web recomanat

Més informació sobre MOFScript en http://eclipse.org/gmt/mofscript/doc/

4.5.1. TCS: *textual concret syntax*

TCS és un llenguatge per a especificar transformacions de model a text que proporciona maneres d'associar elements sintàctics (és a dir, paraules clau com *if*, símbols especials com "+") amb elements del metamodel amb molt poca redundància. Tant les transformacions M2T com T2M poden ser desenvolupades utilitzant una especificació senzilla. Per a fer la transformació se'n necessita l'especificació TCS i el metamodel i model origen.

4.5.2. Cas d'estudi d'M2T

Ara mostrarem la senzillesa de TCS per mitjà d'un cas d'estudi en el qual volem convertir en text models conformes al metamodel de la figura 28. Per exemple, si tenim el model de l'inventari mostrat en la figura 29, el nostre objectiu és definir una transformació `Inventory` que produeixi exactament el text de la figura 35, que ens dóna informació textual sobre l'inventari.

Figura 35. Informació textual sobre l'inventari de la figura 28

```
New Inventory:
  Asset of type HeadGen in second floor
  Asset of type HandleGen in second floor
  Asset of type Assembler in first floor
  Asset of type Polisher in ground floor
  Asset of type Tray in second floor
  Asset of type Tray in second floor
  Asset of type Tray in first floor
  Asset of type Tray in ground floor
  Item: finished Head with cost 5.40 €
  Item: finished Head with cost 5.40 €
  Item: finished Handle with cost 4.35 €
  Item: unfinished Hammer with cost 11.15 €
  Item: finished Hammer with cost 14.80 €
  Item: finished Hammer with cost 14.80 €
End of Inventory
```

Per a generar automàticament aquesta informació podem utilitzar la transformació següent de model a text escrita en TCS:

```
syntax Inventory {
  enumerationTemplate Location auto :
       #groundFloor = "ground floor",
       #firstFloor    = "first floor",
       #secondFloor = "second floor" ;
  enumerationTemplate State auto :
       #finished      = "finished",
       #unfinished  = "unfinished" ;
  template Inventory main :
     "New Inventory:"
     [assets items]{nbNL = 1, indentIncr = 1}
     "End of Inventory" ;
  template Asset :
    "Asset of type " assetType " in " location ;
  template Item :
    "Item: " state " " itemName " with cost " price "€" ;
}
```

Les dues primeres plantilles són introduïdes per a especificar el que cal escriure quan aparegui una dada de tipus enumerat. A l'esquerra de l'assignació s'escriuen els valors del tipus enumerat i a la dreta el que es vol que aparegui en el text.

Consulta recomanada

En l'article "TCS: a DSL for the Specification of Textual Concrete Syntaxes in Model Engineering", de Frédéric Jouault, Jean Bézivin i Ivan Kurtev, s'explica en profunditat TCS. També hi ha més informació a: http://wiki.eclipse.org/index.php/tcs

La plantilla `Inventory` és la que inicia la transformació. Es posa entre cometes el que es vol que aparegui tal qual sempre que ens trobem amb un objecte de tipus `Inventory`. Així, al principi s'escriu "`New Inventory`:". El que apareix en la línia just després, `[assets items]`, es refereix a la relació d'`Inventory` amb `Asset` i amb `Item`, respectivament. Així, després de l'escriptura de "`New Inventory`:" en el fitxer de sortida, es bolcarà el text proporcionat per les plantilles per als objectes de tipus `Asset` i a continuació el de les plantilles associades als objectes de tipus *Item*. El que apareix després, `{nbNL = 1,`

`identIncr = 1}`, fa que la informació de cada objecte s'escrigui en una línia nova i s'apliqui sagnia. Una vegada processats tots els objectes de tipus `Asset` i `Item` s'escriu la cadena "`End of Inventory`".

En les plantilles per als objectes *Asset* i *Item* es pot observar com s'accedeix a la informació dels atributs dels objectes: simplement escrivint el nom de l'atribut. Amb això s'obté una cadena de text que representa el valor de l'atribut. Així, en les dues plantilles s'utilitzen cadenes de text fixes que es concatenen amb els valors recuperats dels atributs.

4.6. Conclusions

Juntament amb els models, les transformacions formen part essencial del desenvolupament de programari dirigit per models i MDA. En aquest capítol hem presentat els conceptes principals de les transformacions de models, els seus mecanismes bàsics, i hem il·lustrat exemples usant dos dels principals llenguatges de transformació de models, com són QVT i ATL.

Encara que aquests llenguatges ja estan bastant madurs i són àmpliament utilitzats, encara necessiten millores en algunes àrees concretes. Per exemple, encara no hi ha depuradors suficientment potents, ni eines d'especificació i proves de transformacions de models. Encara que al principi les transformacions que es necessitaven eren raonablement petites, la seva mida i complexitat ha crescut últimament de manera notable: actualment molts projectes utilitzen transformacions ATL amb centenars de regles i milers de línies de codi. També calen avenços en les teories que suporten els llenguatges de transformació de models per a poder raonar sobre les transformacions i les seves regles, i poder definir mecanismes de modularitat, reutilització, extensió, refactorització, herència, etc. En aquest sentit, la comunitat científica treballa intensament en aquests aspectes, que se solen debatre en el si de conferències especialitzades com la ICMT (International Conference on Model Transformations).

Resum

En aquest mòdul hem presentat els conceptes i mecanismes que s'utilitzen en el desenvolupament de programari dirigit per models, una proposta per al desenvolupament de programari en la qual els models, les transformacions de models i els llenguatges específics de domini tenen els papers principals, enfront de les propostes tradicionals basades en llenguatges de programació, plataformes d'objectes i components de programari.

L'objectiu d'MDD és reduir els costos i temps de desenvolupament de les aplicacions de programari i millorar la qualitat dels sistemes que es construeixen, amb independència de les plataformes d'implementació i garantint les inversions empresarials enfront de la ràpida evolució de la tecnologia.

Malgrat que encara es troba en els seus començaments, MDD ja ha cobrat cos com a disciplina reconeguda i cada vegada més acceptada per la comunitat d'enginyeria del programari per al desenvolupament d'aplicacions i grans sistemes. Per descomptat, MDD no és la panacea que és aplicable a tot tipus de situacions i projectes, i potser un aspecte determinant per a implementar-lo amb èxit és saber determinar en quins projectes, empreses i circumstàncies interessa realment aplicar-lo i en quins no. I per a això és imprescindible conèixer a fons els seus conceptes, fonaments i eines principals; aquest ha estat precisament l'objecte del mòdul present.

Activitats

1. Definiu el metamodel de les comandes en línia de GNU/Linux, que consten d'un nom, zero, un o més arguments i zero, un o més *flags* (arguments precedits de guió). Indiqueu el model corresponent a la comanda: `rm -rf uoc.bak uoc.tmp`, composta pel nom, dos *flags* i dos arguments.

2. Modifiqueu el model de la cadena de muntatge de martells per a tenir en compte que les màquines (generadors, acobladora i polidora) poden produir peces defectuoses. Com es reflecteix això en la resta dels models?

3. Modifiqueu la transformació de models *Plant2Inventory* tenint en compte els canvis fets en els diferents metamodels per a considerar peces defectuoses, segons l'activitat anterior.

4. Especifiqueu en el model de la cadena de muntatge una operació auxiliar OCL que permeti conèixer quina és la safata d'entrada més propera a una màquina. La distància es mesura amb l'operació "`distanceTo()`" especificada en l'apartat "Creació de variables i operacions addicionals". Especifiqueu-ne una altra que retorni la safata més propera, ja sigui d'entrada o de sortida.

5. Definiu un metamodel per a Twitter, amb els seus principals conceptes (*tweet*, usuari, missatge directe, llista, etc.) i les relacions entre aquests (per exemple, els seguidors d'un usuari, els *tags* d'un *tweet*, etc.).

6. Afegiu operacions al metamodel de Twitter desenvolupat en l'activitat anterior (p. ex., publicar un *tweet*, seguir un usuari o respondre un missatge), i especifiqueu-ne el comportament utilitzant OCL.

7. Considereu el model de la figura 13, i definiu quatre metamodels, cadascun representant la sintaxi abstracta de les quatre interpretacions diferents del model esmentades en el text, i compareu-los entre si. Quines són les diferències entre els models?

8. Definiu el *helper* `getPrice` utilitzat en la transformació *Plant2Inventory* (apartat "ATL matched rules") per a assignar preu a les peces.

9. Definiu un metamodel amb la sintaxi abstracta d'un subconjunt mínim del llenguatge Java, que només contingui la definició de classes, els seus atributs i mètodes.

10. Implementeu una transformació de models en ATL que prengui com a entrada un model conforme al metamodel de Java definit en l'exercici anterior i retorni un altre model conforme a aquest mateix metamodel, però en què tots els atributs públics es converteixin en privats i es generin els mètodes *getter* i *setter* corresponents per a aquests.

Exercicis d'autoavaluació

1. Indiqueu les principals diferències entre MDA, MDD i MDE.

2. Descriviu la diferència entre un model i un metamodel. Indiqueu dos llenguatges de modelització i dos de metamodelització.

3. Descriviu els components principals del patró MDA.

4. Descriviu el significat i ús de la paraula reservada `context` en OCL.

5. Assenyaleu la diferència entre els operadors "^" i "^^" d'OCL.

6. Indiqueu els principals components d'un llenguatge específic de domini, i expliqueu com s'especifica cadascun.

7. Descriviu les tres maneres que ofereix OMG per a definir llenguatges específics de domini.

8. Descriviu la diferència entre *tag definition* i *tag value* en la definició d'un perfil UML.

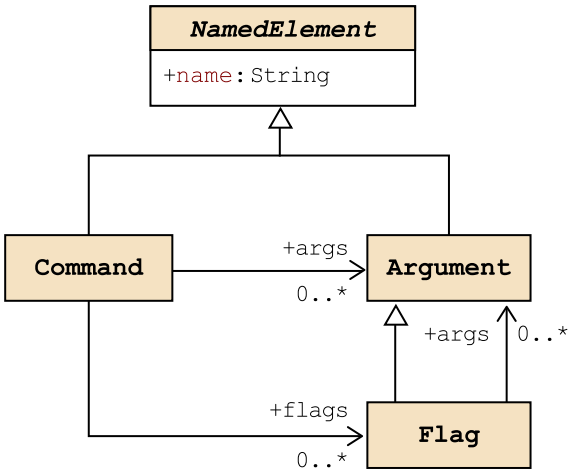
9. Descriviu la diferència entre una transformació de models *in-place* i una que no ho sigui.

10. Descriviu les diferències entre els tres tipus de regles declaratives d'ATL: *matched rules*, *lazy rules* i *unique lazy rules*.

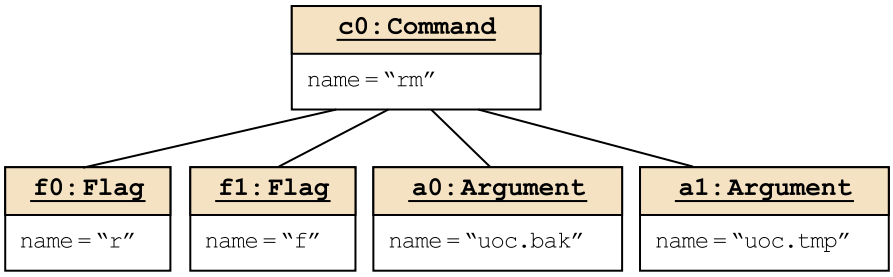
Solucionari

Activitats

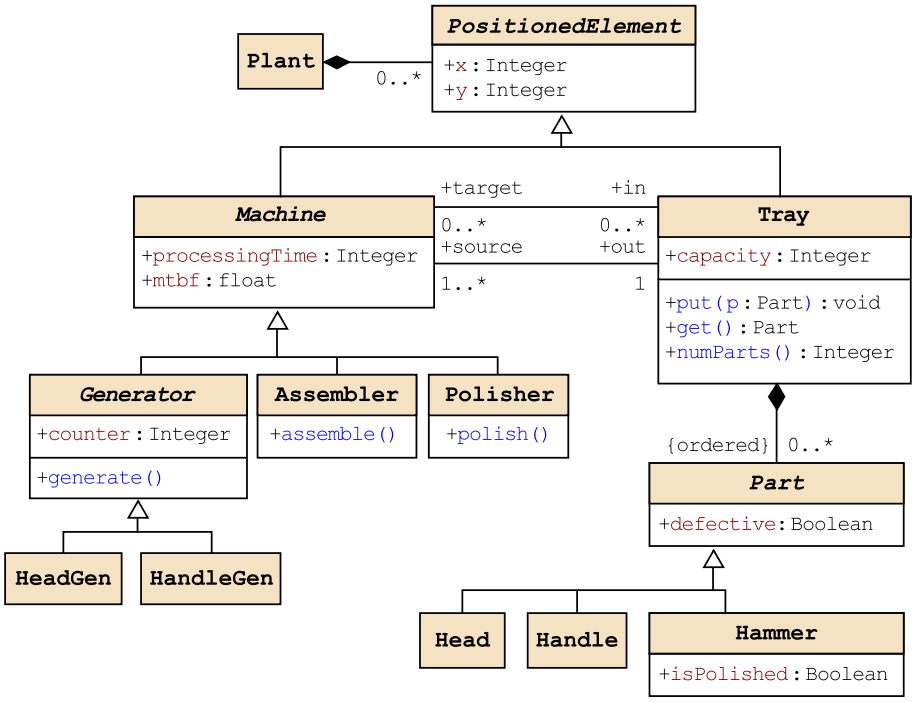
1. El metamodel demanat es pot representar de la manera següent:



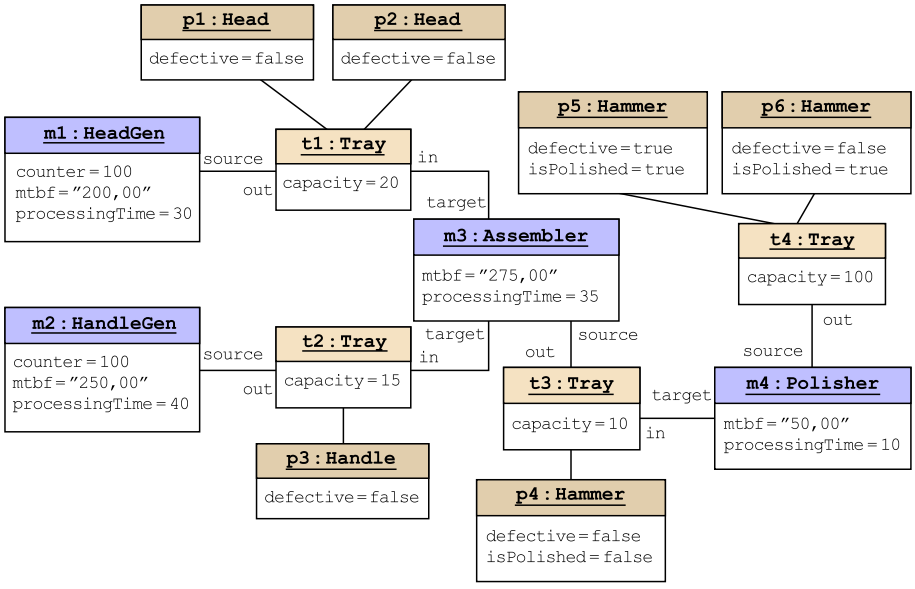
Donat aquest metamodel, el model corresponent a la comanda `rm -rf uoc.bak uoc.tmp` seria el següent:



2. Els nous requisits podrien ser implementats mitjançant la inclusió d'un atribut (*defective : Boolean*) en la classe `Part` i un atribut (*mtbf : Float*) en la classe `Machine`, tal com es pot veure en el metamodel modificat:



L'atribut *defective* emmagatzema la informació sobre si la peça és defectuosa o no, i l'atribut *mtbf* emmagatzema un valor numèric que indica el temps mitjà entre fallades de la màquina (mesurat en segons). Amb aquests canvis, un possible model conforme a aquest metamodel és el mostrat a continuació, en el qual la polidora és ràpida però té una elevada taxa de fallades i, de fet, ha produït una peça defectuosa (*p5*):



3. Per a considerar peces defectuoses, segons l'activitat anterior, considerem que el nou metamodel origen és *PlantModified*, i que el model destinació (*Inventory*) no canvia. Solament no es tenen en compte les peces defectuoses, que són excloses.

```
module PlantModified2Inventory;
create OUT : Inventory  from IN : PlantModified;
rule Machine2Asset {
   from  m : PlantModified!Machine
   to    a : Inventory!Asset(
              assetType <- m.getMachineType(),
              location <- m.y.getFloor()    )
}
rule Tray2Asset { -- does not change
   from  t : PlantModified!Tray
   to    a : Inventory!Asset(
               assetType <- 'Tray',
               location <- t.y.getFloor() )
}
rule Part2Item {
   from  p : PlantModified!Part (not p.defective)
   to    i : Inventory!Item(
               itemName <- p.getItemName(), ),
               state <- p.getPartState(),
               price <- p.getPrice() )
}
```

La resta de la transformació (per exemple, els *helpers*) no canvia.

4. L'operació auxiliar que retorna la safata d'entrada més propera a una màquina es pot especificar com segueix:

```
context Machine
def closestInTray() : Tray
pre:  self.in->notEmpty()
post: result = self.in->any(c : Tray |
        forAll(t : Tray | c.distanceTo(self) <= t.distanceTo(self)))
```

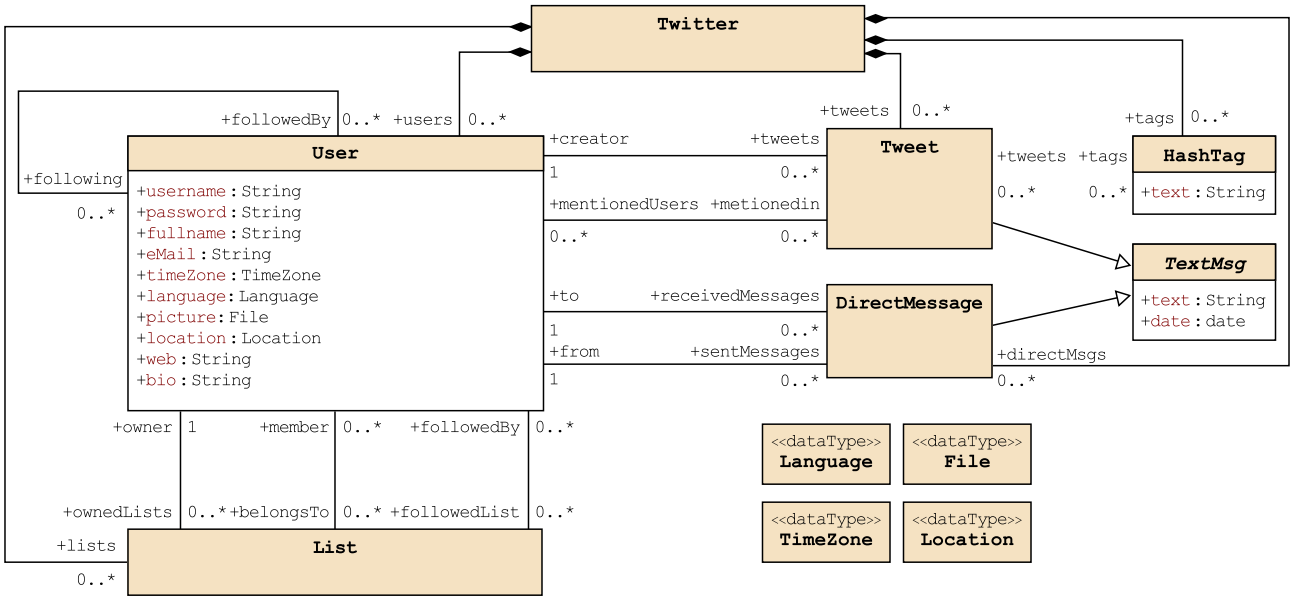
L'operació auxiliar que retorna la safata més propera a una màquina, independentment de si és d'entrada o de sortida, es pot especificar de dues maneres diferents. En primer lloc com:

```
context Machine
def closestTray() : Tray
pre:  self.in->union(self.out)->notEmpty()
post: result = self.in->union(self.out)->any(c : Tray |
        forAll(t : Tray | c.distanceTo(self) <= t.distanceTo(self)))
```

o, alternativament:

```
context Machine
def closestTray() : Tray
pre:  self.in->notEmpty() or self.out->notEmpty()
post: result = Tray.allInstances()->any(c : Tray |
        forAll(t : Tray | c.distanceTo(self) <= t.distanceTo(self)))
```

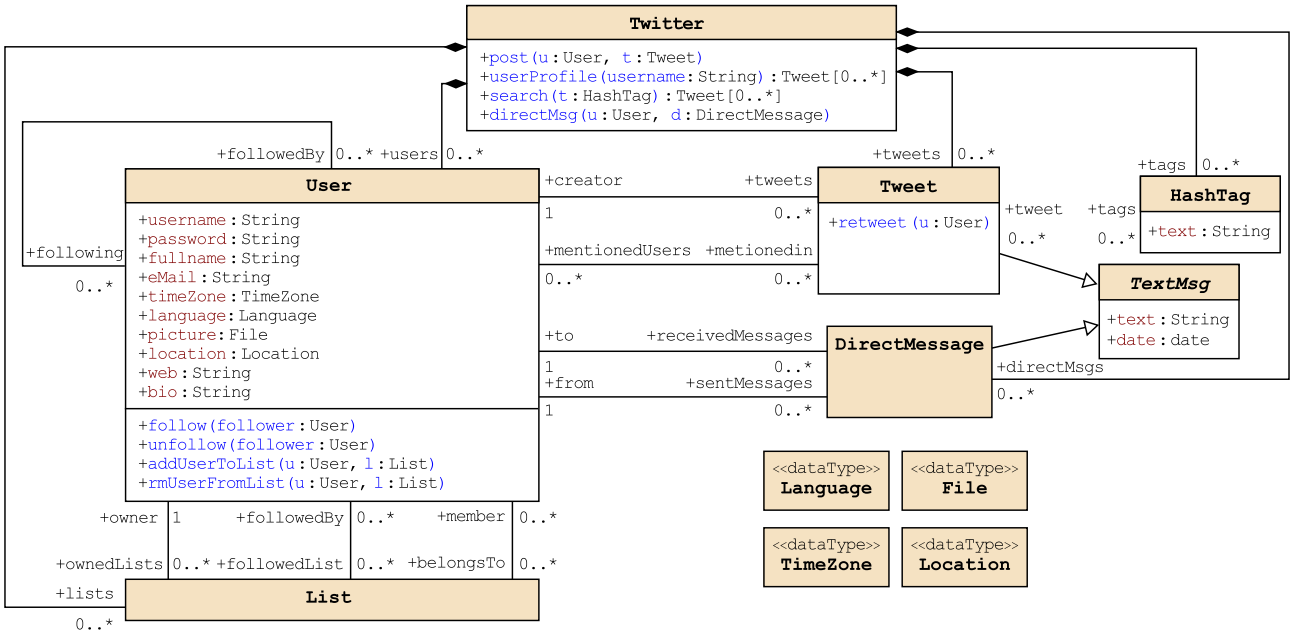
5. Un possible metamodel per a Twitter, descrit per un diagrama UML amb els seus conceptes principals i les relacions entre aquests, és el següent:



Quant a les restriccions d'integritat, podem considerar:

```
context Twitter inv usernameIsKey:
   self.users->isUnique(username)

context User inv noAutoFollow:
   self.following->excludes(self) and self.followedBy->excludes(self)

context TextMsg inv hundredFortyCharsIsEnough:
   self.text.size()<=140

context Tweet inv validMention:
   self.mentionedUser->forAll(u |
            self.text.indexOf("@"+u.username)<>0))

context Tweet inv mentionedTag:
   self.tags->forAll(t | (self.text.indexOf(t.text)<>0))

context HashTag inv validTag:
   self.text.indexOf("#")=1  -- startsWith "#"
   and     -- and does not contain another "#"
   self.text.substring(2,self.text.size()).indexOf("#") = 0
   and     -- and does not contain spaces
   self.text.indexOf(" ") = 0

context DirectMessage inv validMsg:
   -- starts with "D" and then a username
   self.text.indexOf("D @"+self.to.userName)=1
```

6. El metamodel de Twitter pot ser enriquit amb operacions, com mostra el diagrama següent:



Aquestes operacions poden ser especificades en OCL com segueix:

```
context Twitter::post(u:User,t:Tweet)
pre:  t.indexOf("D @") <> 1  -- it is not a direct message
post:    self.tweets->includes(t) and
      u.tweets->includes(t) and
         t.creator = u and
      t.tags->includesAll(t.mentionedTags()) and
      t.mentionedUsers->includesAll(t.mentionedUsers())

context Twitter::directMsg(u:User,d:DirectMsg)
pre: -- user u can send direct messages only to users that follow u
    let uname : String =
         d.substring(4,d.substring(4,d.size()).indexOf(" ")) in
      u.followedBy->any(u | u.username = uname)
post: let uname : String =
         d.substring(4,d.substring(4,d.size()).indexOf(" ")) in
      self.directMsgs->includes(d) and
      d.to = self.users->one(username = uname) and
      d.from = u

-- AUXILIARY OPERATIONS
context Tweet::mentionedTags() : Set{HashTag}
-- returns the set of tags mentioned in a tweet
body: ...
context Tweet::mentionedUsers() : Set{User}
-- returns the set of users mentioned in a tweet
body: ...
```

7. Les possibles interpretacions són les següents:

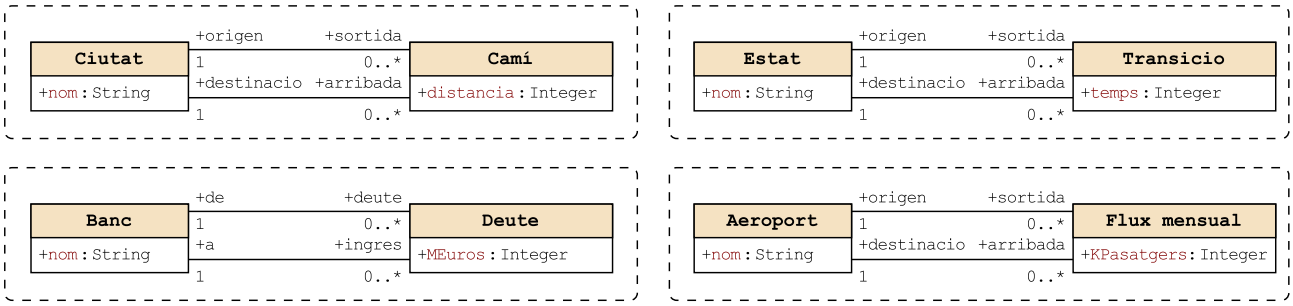
1) Distàncies entre ciutats, expressades en quilòmetres.

2) Transicions entre estats d'un diagrama d'estats, en què els nombres indiquen el temps en mil·lisegons que es triga a dur a terme cada transició.

3) Fluxos de migració mensuals entre aeroports, expressats en milers de persones.

4) Deutes entre entitats bancàries, expressats en milions d'euros.

I els metamodels corresponents són:

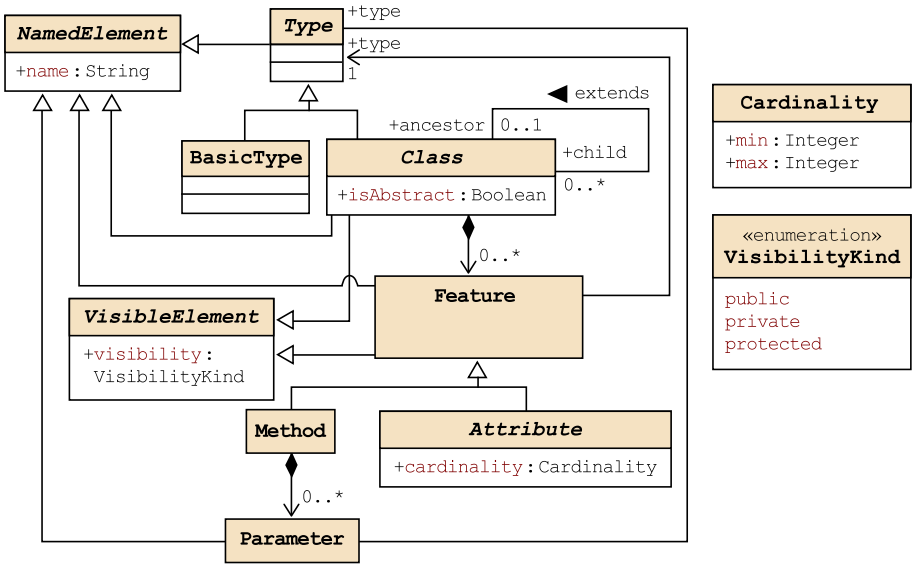


Excepte els noms de les classes i els atributs, els quatre metamodels són els mateixos. El que distingirà uns d'altres és la seva semàntica.

8. Segons es detalla en l'apartat "ATL matched rules", l'atribut *Price* indica el preu de la peça. Els caps de martell tenen un preu de 5,40 €, els mànecs costen 4,35 €, un martell sense polir 11,15 € i un de polit 14,80 €.

```
helper context Plant!Part def : getPrice() : Real =
   if (self.oclIsTypeOf(Plant!Head)) then 5.40
   else if (self.oclIsTypeOf(Plant!Handle)) then 4.35
   else if (self.oclIsTypeOf(Plant!Hammer) and not self.isPolished)
        then 11.15 else 14.80 endif
   endif endif;
```

9. Un metamodel per al subllenguatge de Java de l'activitat podria ser el següent:



10. La transformació ATL demanada és una transformació *in-place*, que només s'ha d'encarregar de definir el seu comportament per als elements que cal transformar, en aquest cas els atributs públics. La resta no cal modificar-los i, per tant, no cal incloure'ls en l'especificació de la transformació.

```
module Public2Private;
create OUT : Java refining IN : Java;
helper context String def: firstToUpper() : String =
   self.substring(1,1).toUpper() + self.substring(2, self.size());
rule PublicAttribute {
   from  s : Java!Attribute ( s.visibility = #public )
   to    t : Java!Attribute ( -- change visibility to #private
            visibility <- #private ),
         tg : Java!Method ( -- define public "getter"
            name <- 'get' + s.name.firstToUpper(),
            visibility <- #public,  type <- s.type ),
         ts : Java!Method (-- define public "setter"
            name <- 'set' + s.name.firstToUpper(),
            visibility <- #public,  parameter <- tsp ),
         tsp : Java!Parameter ( -- parameter of "setter".
            name <- 'newValue', type <- s.type )
}
```

Exercicis d'autoavaluació

1. MDD (*model driven development*) és més específic que MDE (*model-driven engineering*), ja que MDD es refereix només a les activitats de desenvolupament d'aplicacions de programari usant models, mentre que MDE engloba també la resta de les activitats d'enginyeria del programari (manteniment, evolució, anàlisi, etc.). MDA és la proposta concreta de l'OMG per a implementar MDD, usant els estàndards i llenguatges definits per aquesta organització. (Vegeu l'apartat "Terminologia".)

2. Un model és una descripció o representació d'un sistema. Un metamodel és un model d'un llenguatge de modelització, que serveix per a descriure models. MOF i Ecore són exemples de llenguatges de metamodelització. UML i BPMN són exemples de llenguatges de modelització. (Vegeu l'apartat "Conceptes bàsics".)

3. El patró MDA està compost per un model PIM del sistema, que es transforma en un model PSM del mateix sistema, usant una transformació de models. Aquesta transformació pot estar parametritzada per un altre model addicional, que pot ser, per exemple, el model de la plataforma destinació, o bé paràmetres de configuració de l'usuari. (Vegeu l'apartat "Els models com a peces clau d'enginyeria".)

4. En OCL, la paraula reservada `context` s'utilitza per a identificar el context en el qual es defineix l'expressió, és a dir, l'element del model que serveix com a referència a l'expressió OCL, i des d'on es fan les navegacions a altres elements. La paraula reservada `self` s'usa precisament per a referir-se a instàncies d'aquest element. (Vegeu l'apartat "Restriccions d'integritat en OCL".)

5. En OCL, l'operador "^" (*hasSent*) s'utilitza per a avaluar si un missatge o senyal ha estat enviat durant l'execució d'un mètode. L'operador "^^" (*SentMessages*) retorna el conjunt de missatges enviats per un objecte durant l'execució d'un mètode. Tots dos operadors només es poden utilitzar en l'especificació de postcondicions d'operacions (Vegeu l'apartat "Invocació d'operacions i sortides".)

6. Un DSL consta de sintaxi abstracta, sintaxi concreta i semàntica. La primera descriu el vocabulari del llenguatge i les seves regles gramaticals; es descriu mitjançant un metamodel. La sintaxi concreta defineix la notació amb la qual es representen els models vàlids que es poden expressar amb el DSL; es descriu mitjançant una associació entre els conceptes del DSL i un conjunt de símbols o paraules (depenent de si el DSL és visual o textual). La semàntica defineix el significat dels models vàlids. Es pot descriure de maneres diferents: denotacionalment, operacionalment o axiomàticament. (Vegeu l'apartat "Components d'un DSL".)

7. L'OMG defineix tres maneres per a definir DSL: 1) directament a partir de MOF; 2) estenent el metamodel d'UML, però respectant-ne la semàntica (extensió "lleugera", usant perfils UML); i 3) modificant el metamodel d'UML sense respectar-ne necessàriament la semàntica (extensió "pesant"). (Vegeu l'apartat "Maneres d'estendre UML".)

8. Un *tag definition* és un atribut d'un estereotip definit en un perfil UML. Aquest atribut pot prendre valors que s'han de definir en el moment d'aplicar l'estereotip sobre un element UML. Aquests valors es denominen *tag values*. (Vegeu la secció "Els perfils UML".)

9. Una transformació de models *in- place* és un tipus particular de transformació en la qual els metamodels origen i destinació són el mateix. (Vegeu l'apartat "Tipus de transformacions".)

10. Una *matched rule* és una regla ATL que defineix un patró i com s'han de transformar els elements del model origen que siguin conformes a aquest patró en elements del model destinació. S'executen cada vegada que hi hagi una coincidència d'elements en el model origen amb el patró que especifica la regla. Una *lazy rule* és un tipus particular de *matched rule* que només s'executa quan és invocada des d'una altra regla ATL. Finalment, una *unique lazy rule* és un tipus particular de *lazy rule* que no genera nous elements en el model destinació cada vegada que s'executa en ser invocada, sinó que a partir de la seva segona invocació sempre retorna els mateixos elements del model destinació per els mateixos paràmetres d'entrada. (Vegeu l'apartat "ATL: Atlas transformation language".)

Glossari

architecture-driven modernization Proposta de l'OMG per a implementar pràctiques d'enginyeria inversa, usant models.
Sigla **ADM**

Atlas transformation language Llenguatge de transformació de models definit pel grup Atlanmod de l'INRIA, d'àmplia adopció.
Sigla **ATL**

business process modeling Branca del *Model-Based Engineering* que se centra en la modelització dels processos de negoci d'una empresa o organització, de manera independent de les plataformes i les tecnologies utilitzades.
Sigla **BPM**

business process modelling notation Llenguatge de modelització de processos de negoci de l'OMG.
Sigla **BPMN**

domain specific language Llenguatge de modelització que proporciona els conceptes, notacions i mecanismes propis d'un domini en qüestió, semblants als que manegen els experts d'aquest domini, i que permet expressar els models del sistema a un nivell d'abstracció adequat.
Sigla **DSL**

llenguatge específic de domini Vegeu **DSL**.
Sigla **LED**

metamodel *m* Model que especifica els conceptes d'un llenguatge, les relacions entre aquests i les regles estructurals que restringeixen els possibles elements dels models vàlids, i també aquelles combinacions entre elements que respecten les regles semàntiques del domini.

meta-object facility Llenguatge de metamodelització definit per l'OMG.
Sigla **MOF**

model (d'un <x>) *m* Especificació o descripció d'un <x> des d'un punt de vista determinat, expressat en un llenguatge ben definit, i amb un propòsit determinat. En aquesta definició, <x> es pot referir a un sistema existent o imaginari, un llenguatge, un artefacte de programari, etc.

model-based engineering Terme general que engloba els enfocaments dins de l'enginyeria del programari que usen models en algun dels seus processos o activitats: MDE, MDA, MDD, etc. (vegeu la figura 7).
Sigla **MBE**

model-driven architecture Proposta concreta de l'OMG per a implementar MDD, usant les notacions, mecanismes i eines estàndard definits per aquesta organització (MOF, UML, OCL, QVT, XMI, etc.).
Sigla **MDA**

model-driven development Paradigma de desenvolupament de programari que utilitza models per a dissenyar els sistemes a diferents nivells d'abstracció, i seqüències de transformacions de models per a generar uns models a partir d'altres fins a generar el codi final de les aplicacions en les plataformes destinació.
Sigla **MDD**

model-driven engineering Paradigma dins de l'enginyeria del programari que advoca per l'ús dels models i les transformacions entre elles com a peces clau per a dirigir totes les activitats relacionades amb l'enginyeria del programari.
Sigla **MDE**

model-driven interoperability Iniciativa per a implementar mecanismes d'interoperabilitat entre serveis, aplicacions i sistemes usant models i tècniques d'MBE.
Sigla **MDI**

model-to-model Tipus particular de transformacions, en les quals l'origen i la destinació són models.
Sigla **M2M**

model-to-text Tipus particular de transformacions, en les quals l'origen és un model i la destinació és un text (per exemple, un tros de codi).

Sigla **M2T**

object constraint language Llenguatge textual d'especificació de restriccions sobre models, definit per l'OMG.
Sigla **OCL**

Object Management Group Consorci per a l'estandardització de llenguatges, models i sistemes per al desenvolupament d'aplicacions distribuïdes i la seva interoperabilitat.
Sigla **OMG**

perfil UML *m* Extensió d'un subconjunt d'UML orientat a un domini, que utilitza estereotips, valors etiquetats i restriccions.

plataforma *f* Conjunt de subsistemes i tecnologies que descriuen la funcionalitat d'una aplicació sobre la qual es construiran sistemes, per mitjà d'interfícies i patrons específics.

platform independent model Model d'un sistema que concreta els seus requisits funcionals en termes de conceptes del domini i que és independent de qualsevol plataforma.
Sigla **PIM**

platform specific model Model d'un sistema resultat de refinar un model PIM per a adaptar-lo als serveis i mecanismes oferts per una plataforma concreta.
Sigla **PSM**

query-view-transformation Llenguatge de transformació de models definit per l'OMG.
Sigla **QVT**

semàntica (d'un DSL) *f* Especificació del significat dels models vàlids que es poden representar amb un DSL.

sintaxi abstracta (d'un DSL) *f* Descriu el vocabulari amb els conceptes del llenguatge, les relacions entre aquests, i les regles que permeten construir les sentències (programes, instruccions, expressions o models) vàlides del llenguatge.

sintaxi concreta (d'un DSL) *f* Defineix la notació que s'usa per a representar els models que es poden descriure amb aquest llenguatge.

systems process engineering metamodel Llenguatge de modelització de processos de negoci de l'OMG.
Sigla **SPEM**

transformació de models *f* Procés de convertir un model d'un sistema en un model del mateix sistema. Així mateix, s'anomena així l'especificació d'aquest procés.

unified modeling language Llenguatge de modelització de propòsit general definit per l'OMG.
Sigla **UML**

XML metadata interchange Estàndard de l'OMG per a l'intercanvi de metadades.
Sigla **XMI**

Bibliografia

Bibliografia bàsica

Clark, T.; Sammut, P.; Willans, J. (2004). *Applied Metamodeling: A Foundation for Language Driven Development* (2a. ed.). Ceteva. http://itcentre.tvu.ac.uk/~clark/docs/Applied%20Metamodelling%20%28Second%20Edition%29.pdf.

Warmer, J.; Kleppe, A. (2003). *The Object Constraint Language: Getting Your Models Ready for MDA* (2a. ed.). Addison Wesley.

Bibliografia addicional

Cook, S.; Jones, S.; Kent, S.; Wills, A. C. (2007). *Domain-Specific Development with Visual Studio DSL Tools*. Willey.

Fowler, M. (2011). *Domain Specific Languages*. Addison Wesley.

Kelly, S.; Tolvanen, J. P. (2008). *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley-IEEE Computer Society Press.

Kleppe, A. (2008). *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison Wesley.

Referències bibliogràfiques

Bézivin, J. (2005). "The Unification Power of Models". *SoSym* (vol. 4, núm. 2, pàg. 171-188).

Jouault, F.; Allilaire, J.; Bézivin, I.; Kurtev (2008). "ATL: a model transformation tool". *Science of Computer Programming* (vol. 72, núm. 1-2, pàg. 31-39).

Mernik, M.; Heering, J.; Sloane, A. (2005). "When and How to Develop Domain-Specific Languages". *ACM Computing Surveys* (vol. 37, núm. 4, pàg. 316-344).

Sendall, S.; Kozaczynski, W. (2003). "Model Transformations: The heart and soul of Model-driven Software Development". *Proceedings of IEEE Software* (vol. 20, núm. 5, pàg. 42-50).

Strembeck, M.; Zdun, U. (2009). "An Approach for the Systematic Development of Domain-Specific Languages". *Software: Practice and Experience (SP&E)* (vol. 39, núm. 15).

Enllaços recomanats

OMG (2001). "Model Driven Architecture – A technical perspective". OMG document: ormsc/2001-07-0.

OMG (2003). "MDA Guide V1.0.1". OMG document: omg/03-06-01.

OMG (2004). "Human-Usable Textual Notation (HUTN) Specification". OMG document: formal/04-08-01.

OMG (2008). "MOF QVT Final Adopted Specification". OMG document: formal/08-04-03.

OMG (2010). "UML 2.3.1 Superstructure specification". OMG document: formal/2010-05-05.

OMG (2012). "OCL 2.3.1". OMG document: formal/2012-01-01.